Procesamiento de imágenes en vehículos autónomos sobre un RISC-V con acelerador
Image processing in autonomous vehicles on a RISC-V with accelerator

Trabajo de Fin de Máster
Curso 2021–2022

Autora
María José Belda Beneyto

Directores
Katzalin Olcoz Herrero
Fernando Castro Rodríguez

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

Procesamiento de imágenes en vehículos autónomos sobre un RISC-V con acelerador
Image processing in autonomous vehicles on a RISC-V with accelerator

Trabajo de Fin de Máster en Internet de las Cosas
Departamento de Arquitectura de Computadores Y Automática

Autora
María José Belda Beneyto

Directores
Katzalin Olcoz Herrero
Fernando Castro Rodríguez

Convocatoria: *Enero 2022* **Calificación:** *9.5*

Máster en Internet de las Cosas
Facultad de Informática
Universidad Complutense de Madrid

7 de febrero de 2022

Autorización de difusión

El abajo firmante, matriculado en el Máster en Internet de las Cosas de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Procesamiento de imágenes en vehículos autónomos sobre un RISC-V con acelerador", realizado durante el curso académico 2021-2022 bajo la dirección de Katzalin Olcoz Herrero y Fernando Castro Rodríguez en el Departamento de Arquitectura de Computadores Y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

María José Belda Beneyto

7 de febrero de 2022

Agradecimientos

Este trabajo no habría sido posible llevarlo a cabo exitosamente sin la inestimable ayuda de las personas citadas a continuación.

En primer lugar, agradecer a Katzalin y Fernando, mis directores, todo el esfuerzo y las horas que han dedicado a resolver los obstáculos en forma de errores y falta de información con los que hemos topado. Así como los ánimos que siempre estaban dispuestos a darme. También agradecer al profesor Carlos García Sánchez su aportación de los códigos de tratamiento de imágenes y su paciencia al explicarnos su funcionamiento y contestar nuestras dudas y peticiones.

En segundo lugar, a mis padres y mi pareja, piezas fundamentales de mi día a día y mi bienestar, y apoyos a los que recurrir cuando no veía la luz al final del túnel. También, a mi familia y amigos de siempre, que por más kilómetros que nos separen siempre nos reencontramos cada poco tiempo para darme ánimos y ganas de seguir en la capital.

Finalmente, agradecer a los amigos que me ha dado mi trayectoria universitaria que son un ejemplo a seguir y siempre tienen un ratito en las noches duras para dejarme ganar al Catán o hacerme reír encontrando huecos “Aquí, donde estoy yo”.

Resumen

Procesamiento de imágenes en vehículos autónomos sobre un RISC-V con acelerador

La conducción autónoma en vehículos terrestres es un campo emergente en el que el Internet de las Cosas (IoT) juega un papel importante en la actualidad. El correcto funcionamiento de este tipo de conducción sin intervención humana requiere el continuo procesamiento de las imágenes que capta el vehículo durante su viaje con la finalidad de determinar de modo preciso la trayectoria a seguir así como la posible presencia de obstáculos en la misma. Dada la elevada cantidad de datos a procesar y la necesaria respuesta del vehículo en tiempo real, es necesaria la exploración de nuevas arquitecturas que permitan la realización de estas tareas de un modo eficiente tanto en términos de latencia como de consumo energético. En este trabajo se propone el diseño y se realiza la evaluación de algunas arquitecturas para este fin en las que se hace uso de procesadores que utilizan el repertorio de instrucciones RISC-V así como de aceleradores y coprocesadores de propósito específico destinados a optimizar la realización de las tareas propias del vehículo autónomo. Para ello se hace uso de una amplia colección de herramientas, como Chipyard, FireSim FireMarshal y Amazon Web Services, necesarias para generar los diseños hardware específicos y ejecutar sobre ellos una aplicación de reconocimiento de líneas en imágenes como las empleadas en los vehículos de conducción autónoma.

Palabras clave

Conducción autónoma, detección de líneas, acelerador, coprocesador, RISC-V, IoT, Hwacha, Gemmini, FireSim

Abstract

Image processing in autonomous vehicles on a RISC-V with accelerator

Currently, autonomous driving in land vehicles is an emerging research field where Internet of Things (IoT) plays a significant role. The right operation of this kind of driving – where no human intervention exists–, requires the continuous processing of the images that the vehicle collects as it travels, aimed to accurately determine the trajectory it must follow as well as the potential presence of obstacles. Given the huge amount of data to be processed and the real-time response required in the vehicle, the exploration of new architectures that allow the implementation of these tasks in an efficient fashion, both in terms of latency and energy consumption, is nowadays an open challenge. In this work several architectures, oriented to these goals, are proposed and evaluated. These architectures employ RISC-V processors and also specific-domain accelerators and coprocessors, aimed to optimize the tasks involving the autonomous vehicle. For this purpose, a wide toolset is used, including Chipyard, FireSim FireMarshal and Amazon Web Services. All of them make it possible to generate specific hardware designs and to execute a line-detection application as that employed in autonomous vehicles.

Keywords

Autonomous driving, line detection, accelerator, coprocessor, RISC-V, IoT, Hwacha, Gemmini, FireSim

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se va a realizar una pequeña introducción al contexto en el que se desarrolla este trabajo y las motivaciones que nos llevan a realizarlo.

En la era tecnológica en la que vivimos, nos esforzamos cada día en conseguir que todas las tareas habituales sean lo más automáticas posible para dedicar nuestros esfuerzos a otras actividades. Es por esto que surge el Internet de las Cosas (IoT), pues necesitamos nuevas tecnologías para diseñar estos sistemas, habitualmente empotrados, que necesitan un equilibrio entre el bajo consumo y el alto rendimiento.

Algunas áreas han sido ya ampliamente estudiadas, como la automatización de las cadenas de producción o la domótica, pues ya es habitual encontrar casas completamente domotizadas que nos facilitan el día a día. Sin embargo, en el sector de transporte terrestre aún no contamos con estas novedades.

En este ámbito, el futuro parece indicar que también tendremos un modelo de transporte inteligente, conectado y autónomo, que haga uso de la inteligencia artificial. Si ahondamos en la tecnología que se utilizará, nos damos cuenta de que reúne diversas áreas. Por un lado, se necesitará una gran potencia de cómputo para ejecutar muchas operaciones simultáneas que se solapan en la conducción, por ejemplo la detección de las líneas de la carretera para determinar la trayectoria y la detección de obstáculos u otros vehículos, así como el reconocimiento de señales de tráfico.

Estas acciones son críticas para la seguridad en la conducción, por tanto, además de necesitar una gran capacidad de cómputo, necesitan ejecutarse a bordo, pues aún con una buena conexión a la red, la latencia sería demasiado grande. Sin embargo, otras operaciones como el seguimiento GPS y la extracción de datos de tráfico a partir del mismo, sí que se pueden enviar y procesar en la nube, ya que no son críticas o no requieren de inmediatez.

Dada la cantidad de operaciones críticas y simultáneas que se deben ejecutar para obtener un buen sistema de conducción autónoma, necesitamos diseñar SoCs empotrados eficientes y seguros [6] con distintos procesadores y aceleradores de carácter específico integrados en un mismo sistema.

1.1. Motivación

Este trabajo se encuadra en la intersección entre el ámbito IoT y el ámbito de los vehículos autónomos. En particular, los vehículos autónomos necesitan ser capaces de reconocer las líneas que delimitan los carriles en las calzadas para calcular la trayectoria deseada e implementar la conducción autónoma.

Esto se traduce en un problema de detección de líneas en imágenes que habitualmente se resuelve con redes neuronales. Ahora bien, las redes neuronales se basan en aplicar una matriz de pesos a una matriz de datos, es decir, en la multiplicación de matrices. Las multiplicaciones de matrices son conocidas por su costosa ejecución, aún mayor cuándo trabajamos con imágenes de alta resolución, ya que nos ofrecen una gran cantidad de píxeles.

Por tanto, en este trabajo, se pretende resolver el problema de la gran cantidad de cómputo desde un punto de vista arquitectónico. En particular, se van a diseñar distintas arquitecturas homogéneas y heterogéneas compuestas por los procesadores de ámbito general *Rocket* y *Boom* y los aceleradores *Gemmini* y *Hwacha* de propósito específico para la multiplicación de matrices y para el procesamiento vectorial, respectivamente.

Esta arquitectura estará diseñada para optimizar la ejecución de una aplicación de reconocimiento de líneas en imágenes, cuya ejecución en un escenario real sería a bordo de un vehículo autónomo. En esta ocasión, la aplicación hará uso de algoritmos conocidos para el procesamiento de imágenes en lugar de implementar una red neuronal para simplificar el código utilizado.

Tanto los procesadores como el acelerador y el coprocesador vectorial elegidos utilizan el repertorio RISC-V junto con algunas instrucciones personalizadas para el acelerador y el coprocesador. Se eligen ambos por ser RISC-V un repertorio de código libre, así como las herramientas que se encuentran a disposición para generar arquitecturas y realizar simulaciones. Además, las herramientas de generación de arquitecturas como Chipyard o FireSim que se van a utilizar, permiten modificar componentes de la arquitectura en detalle de forma sencilla y cómoda. Así, se pueden añadir o eliminar módulos para adaptar la arquitectura el máximo posible al caso de uso y conseguir el deseado equilibrio entre rendimiento y consumo que se persigue en IoT.

1.2. Objetivos

A continuación, vamos a describir los objetivos que se persiguen con este trabajo. En primer lugar, se persigue explorar el ecosistema RISC-V, tanto el repertorio de instrucciones como los cores, las opciones de hardware de propósito específico y las herramientas software para generar diseños hardware que se encuentran a disposición del usuario.

Del los primeros objetivo se desprende la necesidad de explorar las arquitecturas heterogéneas, para familiarizarnos con el uso de las instrucciones específicas que se añaden al repertorio RISC-V con algunos módulos y para poder conectar un procesador y un hardware de propósito específico en un mismo diseño hardware.

También, se persigue relacionarnos con el mundo de los vehículos autónomos, en particular, con el tratamiento de imágenes dad su importancia en dicho contexto. En este

trabajo, nos centraremos en la detección de líneas en imágenes, ya que es una tecnología cada vez más presente en los vehículos actuales así como en los prototipos de vehículos autónomos. Para ello, nos familiarizaremos con los algoritmos de tratamiento de imágenes que se utilizan en este escenario: el algoritmo de Canny y la transformada de Hough.

Todo esto nos lleva a cumplir nuestro objetivo final, que es conseguir una plataforma en la que podamos ejecutar aplicaciones personalizadas sobre arquitecturas diseñadas adhoc para dichas aplicaciones y conseguir en la ejecución de estas aplicaciones un gran rendimiento a un bajo coste energético, ya que la tecnología embarcada en un vehículo autónomo debe ser rápida pero barata en términos energéticos.

1.3. Plan de trabajo

El plan de trabajo que vamos a seguir para completar los objetivos que se han planteado empieza por realizar un primer acercamiento al código de tratamiento de imágenes y detectar pequeñas mejoras que se puedan aplicar de forma sencilla, como utilizar las variables de tamaño más pequeño que se adapten a la situación. A continuación, se realizará un estudio del rendimiento del código de tratamiento de imágenes mediante un *profiling* para averiguar en qué partes se consume un mayor tiempo y, por tanto, más recursos energéticos también. Así, sabremos qué operaciones debemos ejecutar a nivel hardware en la arquitectura.

A continuación, realizaremos una exhaustiva investigación de los procesadores, coprocesadores y aceleradores que actualmente implementan el repertorio de instrucciones RISC-V y son de código libre y abierto, para elegir aquellos que se adapten mejor a las necesidades de nuestra aplicación.

Una vez decidido el hardware necesario, utilizaremos los entornos de desarrollo Chipyard y FireSim para diseñar las arquitecturas que creamos más adecuadas y, posteriormente, generar imágenes de esas arquitecturas para volcarlas sobre una FPGA. Además, compilaremos el código de nuestra aplicación principal de detección de líneas utilizando el compilador cruzado de RISC-V y otras herramientas a nuestro alcance para finalmente, simular la ejecución del programa principal de detección de líneas sobre dichas arquitecturas.

Finalmente, discutiremos los resultados obtenidos en las ejecuciones de distintas aplicaciones sobre distintas arquitecturas con el objetivo de concluir qué arquitectura se adapta mejor a nuestro caso de uso.

A continuación, en el siguiente capítulo de este trabajo se detallarán las características principales tanto del hardware elegido como de los entornos de desarrollo que se van a utilizar. En capítulos sucesivos, se presentarán los algoritmos de procesamiento utilizados, y los resultados obtenidos. El trabajo finaliza con la exposición de las conclusiones derivadas de la realización del mismo.

Capítulo 2

Estado de la Cuestión

En este capítulo vamos a presentar el repertorio de instrucciones RISC-V y los elementos hardware que utilizaremos en este trabajo, entre los que se incluyen los procesadores Rocket y Boom así como el acelerador Gemmini y el coprocesador Hwacha. A continuación, se presentan las herramientas empleadas en el desarrollo de este proyecto, como Chipyard, FireSim FireMarshal y Amazon Web Services, necesarias para generar los diseños hardware específicos y ejecutar sobre ellos una aplicación de reconocimiento de líneas en imágenes.

2.1. RISC-V

RISC-V es un repertorio de instrucciones de código libre y abierto, al contrario que la mayoría de repertorios, basado en un conjunto de instrucciones reducido (RISC), fruto de un proyecto en la Universidad de California en Berkeley, que empezó en el año 2010 con las aspiraciones de generar un repertorio de instrucciones para docencia [35]. Sin embargo, el proyecto generó curiosidad y tuvo éxito, de forma que, en la actualidad, el repertorio RISC-V es un repertorio completo y funcional que está en continuo desarrollo dada la gran comunidad de contribuyentes con la que cuenta y es mantenido por la fundación "RISC-V Foundation".

Actualmente, existen distintos repertorios RISC-V básicos que se diferencian en el tamaño de los registros de enteros. Estos son RV32I y RV64I con enteros de 32 y de 64 bits respectivamente. Además, existen las variantes RV32E y RV64E análogas a las anteriores pero con la mitad de registros de enteros, pensadas especialmente para microcontroladores que requieran pocas prestaciones y cuenten con un espacio limitado.

Vistos estos cuatro repertorios base, a cualquiera de ellos se les pueden añadir extensiones estándar y específicas para generar repertorios adaptados al caso de uso. Las más comunes son:

- M: extensión estándar para instrucciones de multiplicación y división de enteros.
- A: extensión estándar para instrucciones atómicas.
- F: extensión estándar para instrucciones en punto flotante de precisión simple.

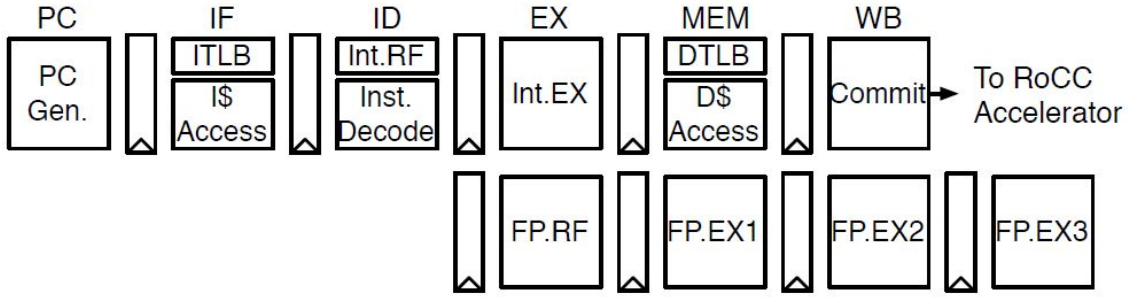


Figura 2.1: Pipeline del core Rocket [3].

- D: extensión estándar para instrucciones en punto flotante de doble precisión.

Estas extensiones junto con el repertorio base (IMAFD) se agrupan y se denotan con la opción "G" [35]. En este trabajo se utiliza el repertorio RV64GC que es el que implementan los cores y aceleradores que utilizaremos. La opción "C" proviene de *Compressed Instructions* y añade instrucciones especialmente cortas, de tan solo 16 bits de codificación, para las operaciones más comunes y, por tanto, permite generar un código ensamblador más corto [30].

2.1.1. Cores

En la actualidad son numerosas las empresas e instituciones públicas que diseñan y producen cores que implementen el repertorio de instrucciones RISC-V. Entre ellos se encuentran los cores Rocket y Boom, diseñados por la universidad de Berkeley y producidos por SiFive. Los motivos por los que se han elegido estos dos cores en el desarrollo de este trabajo es que son de código libre y abierto y están parametrizados, de forma que es sencillo modificar su arquitectura. A continuación, vamos a ver en detalle las características de ambos cores.

2.1.1.1. Rocket

El core Rocket es un procesador implementado en el lenguaje hardware Chisel (Constructing Hardware In a Scala Embedded Language) [8], compuesto por un pipeline de seis etapas que ejecuta instrucciones en orden. Las etapas se presentan en la Figura 2.1 y son las siguientes: generación del PC; Fetch, donde lee la siguiente instrucción de la caché de instrucciones; Decode, donde averigua de qué tipo es la instrucción leída y qué registros necesita como operandos para determinar posibles bloqueos; Execution, donde realiza la operación asociada a la instrucción; Memory, donde accede a la caché de datos; y Commit o Write Back, donde actualiza el estado de la arquitectura con los cambios generados por la instrucción.

A continuación, en la Figura 2.2 vemos la microarquitectura del core Rocket. A pesar de ser un core de rendimiento limitado, dispone de algunas mejoras como un predictor de saltos o un prefetcher, este último destinado a aumentar la eficiencia en la búsqueda de datos en memoria.

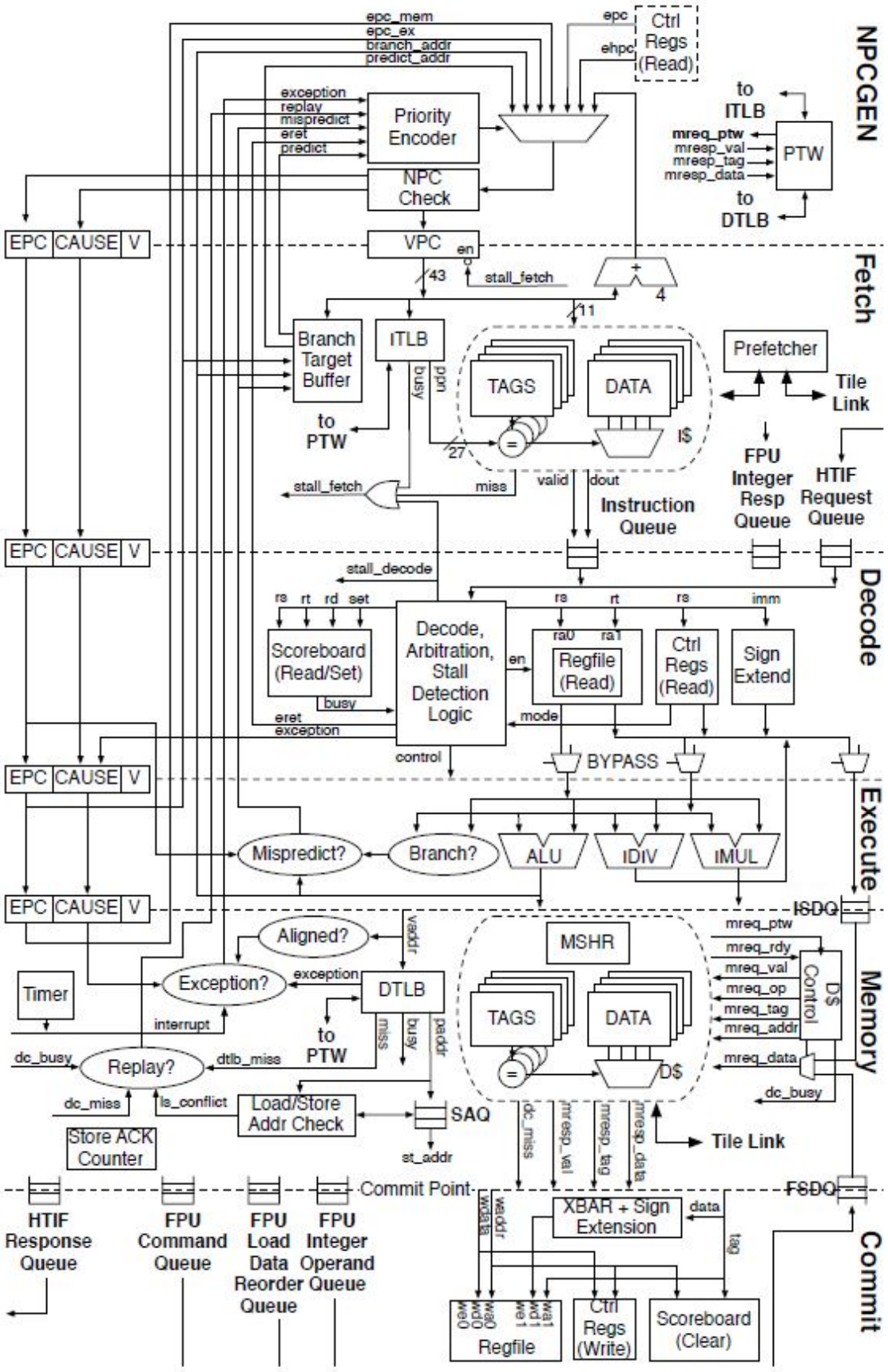


Figura 2.2: Microarquitectura del core Rocket [36].

2.1.1.2. Boom

El core Boom (Berkeley Out-of-Order Machine) es un procesador implementado en Chisel con un pipeline de diez etapas que ejecuta instrucciones fuera de orden. Las etapas detalladas se presentan en la Figura 2.3.

Actualmente, el core Boom cuenta con distintas versiones, cada una mejora de la anterior. En la Figura 2.4 vemos los rasgos generales del pipeline de cada una de ellas. La historia de BOOM [39] empieza con una primera versión BOOMv1, que se inspiró en los procesadores MIPS R10000 [37] y Alpha 21264 [19] y fue creada con la intención de usarse para docencia. Esta primera versión contaba con un pipeline muy simple de pocas etapas, que era físicamente imposible producir, así pues, con la siguiente versión BOOMv2 se buscó un diseño apto para producción. Algunos de los cambios apreciables entre versiones son el aumento del número de etapas en el pipeline y la separación de los registros y las unidades aritméticas para operaciones en punto flotante. Finalmente, la versión más reciente BOOMv3 presenta un mayor número de etapas en el pipeline para evitar cuellos de botella y un diseño específico para explotar la ejecución de instrucciones fuera de orden.

Dada la complejidad de las versiones BOOMv2 y BOOMv3, en la Figura 2.3 podemos ver en detalle cada etapa del pipeline. En este trabajo usaremos la version BOOMv2 que cuenta con un pipeline de diez etapas y la suficiente complejidad arquitectónica para obtener beneficios en términos de rapidez y eficacia respecto al core Rocket, tal y como se establece en [20] y [29], y como posteriormente comprobaremos en este trabajo.

2.1.2. Aceleradores y coprocesadores

El repertorio de instrucciones RISC-V cuenta con numerosos aceleradores de propósito específico, como el SHA3 [33] para encriptación o el Gemmini [15] para multiplicación de matrices, así como coprocesadores como el Hwacha para operaciones vectoriales [27]. A continuación, veremos en detalle el acelerador Gemmini y el coprocesador vectorial Hwacha que usaremos posteriormente en este trabajo.

2.1.2.1. Hwacha

El coprocesador vectorial Hwacha está diseñado para ofrecer un alto rendimiento con un bajo consumo de energía [27]. Su diseño está inspirado en los primeros procesadores vectoriales como el Cray-1 [32] y en los anteriores componentes vectoriales diseñados para RISC-V como Scale [23] [24] o Maven [5] [25] [26]. La principal característica de Hwacha, que lo diferencia del resto de coprocesadores vectoriales del panorama actual, es el alto desacoplamiento entre el acceso a los datos vectoriales y la ejecución vectorial, pues a las operaciones vectoriales se les invoca desde el bucle de procesamiento por bloques y se ejecutan en paralelo al resto del bucle. Con esto, se consigue un gran rendimiento, dando lugar a un paradigma en el que los coprocesadores vectoriales de dimensiones relativamente pequeñas permiten sustituir a las GPUs en algunos casos de uso [9].

El coprocesador cuenta con 256 registros vectoriales configurables, 16 registros de predicado vectoriales también configurables, 32 registros escalares de dirección fijos de solo lectura y 64 registros escalares fijos de escritura y lectura, como podemos ver en la Figura 2.5.

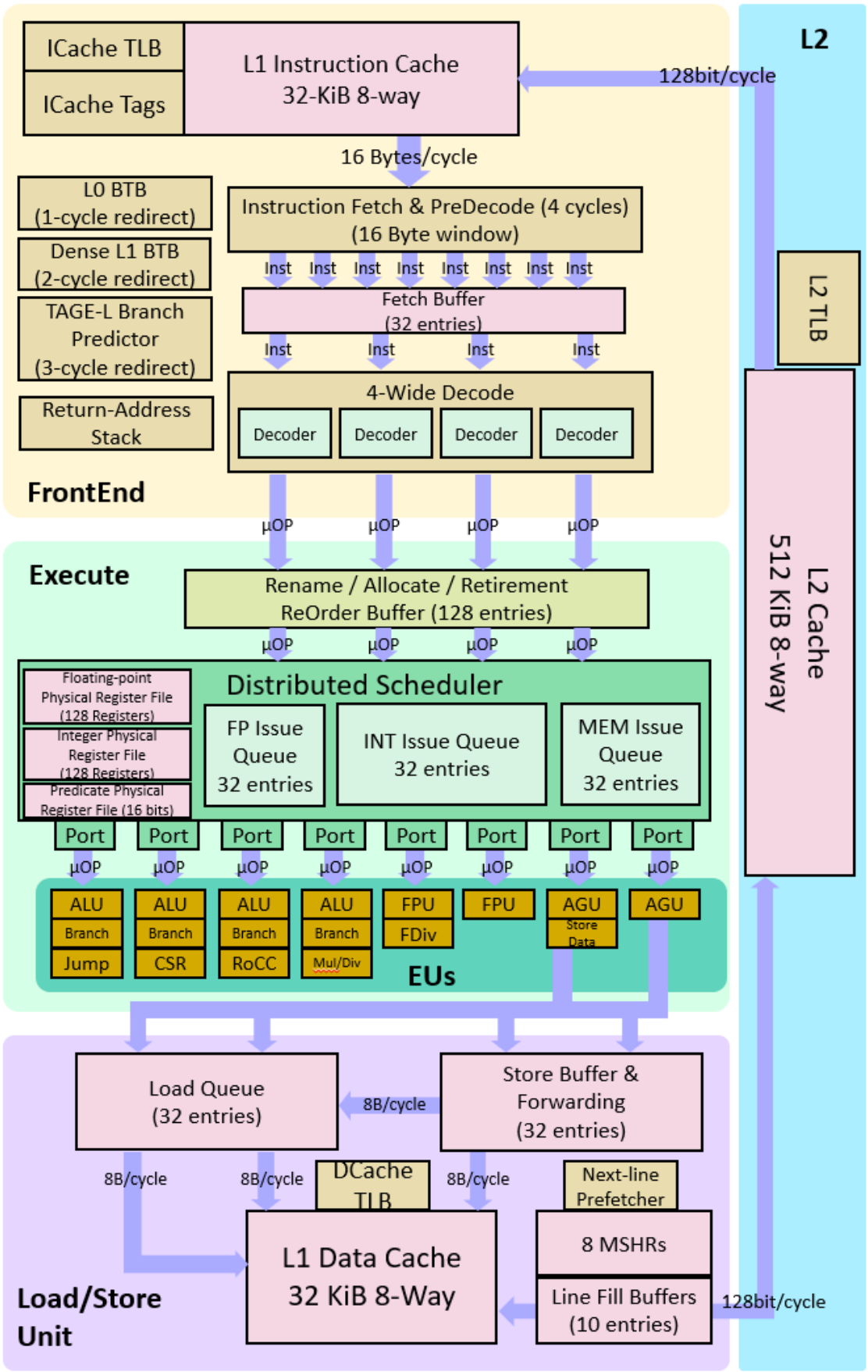


Figura 2.3: Pipeline detallado del core Boom. [38]

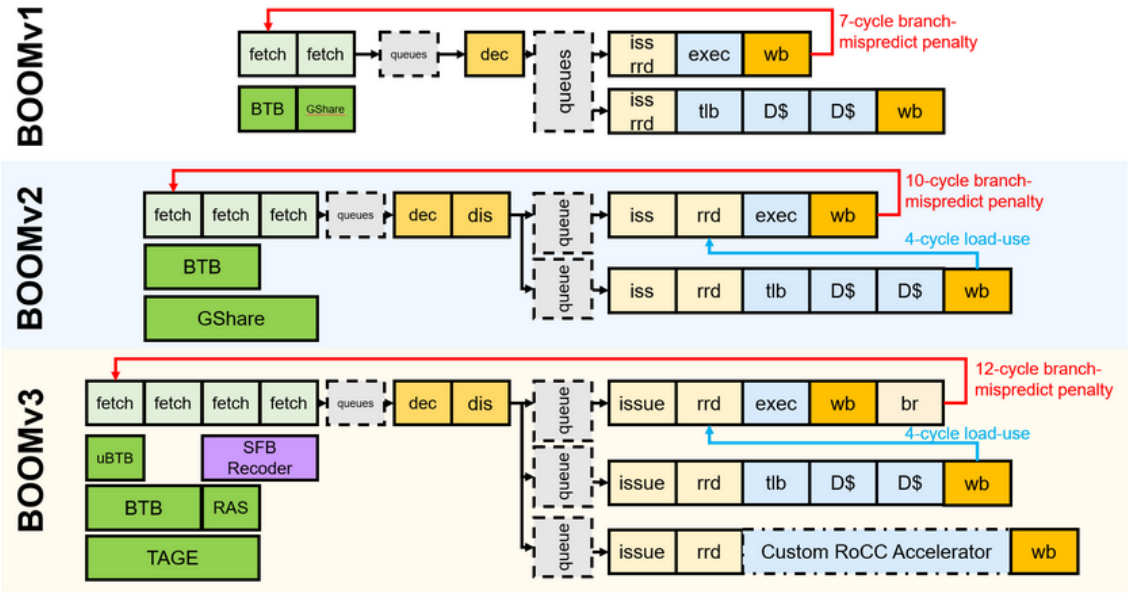


Figura 2.4: Pipeline del core Boom en sus distintas versiones [38].

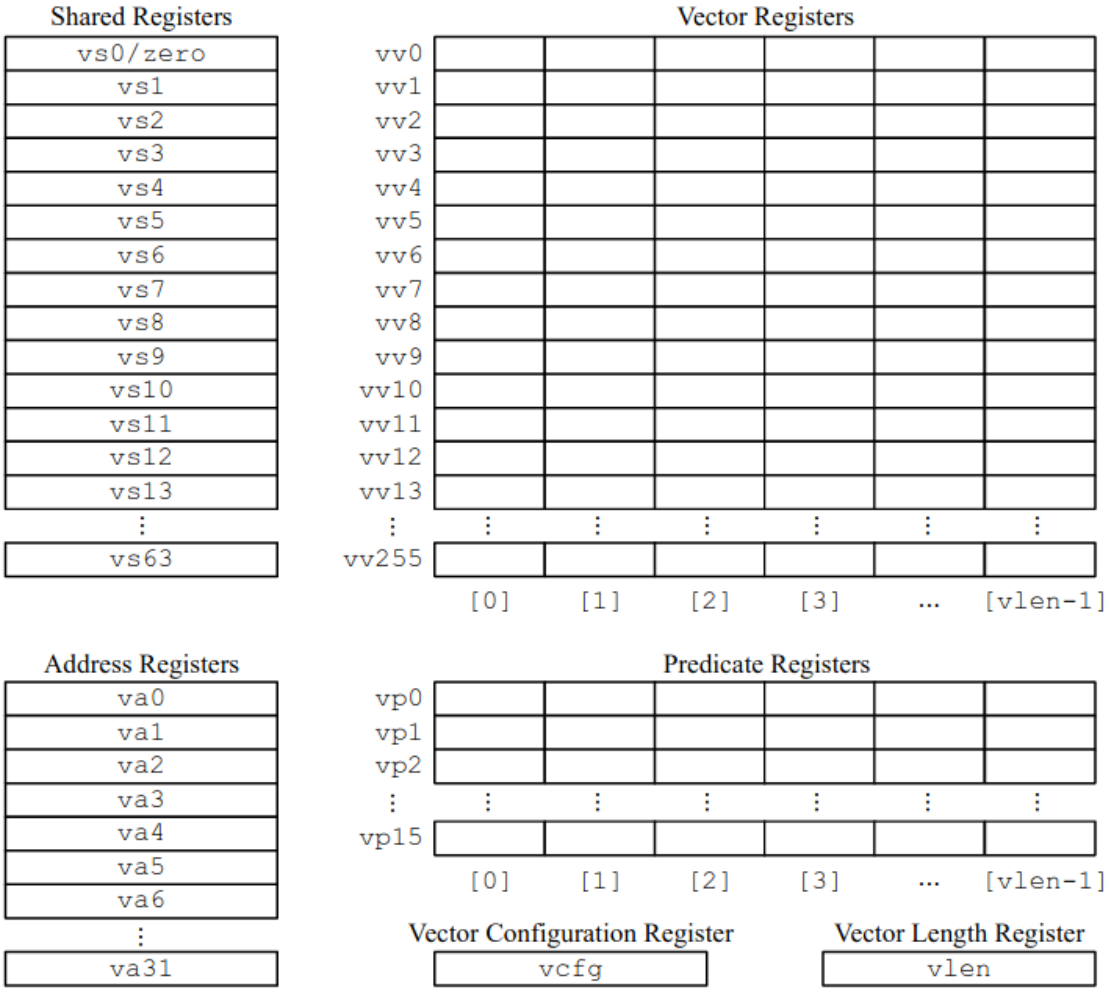


Figura 2.5: Registros del coprocesador Hwacha [27].

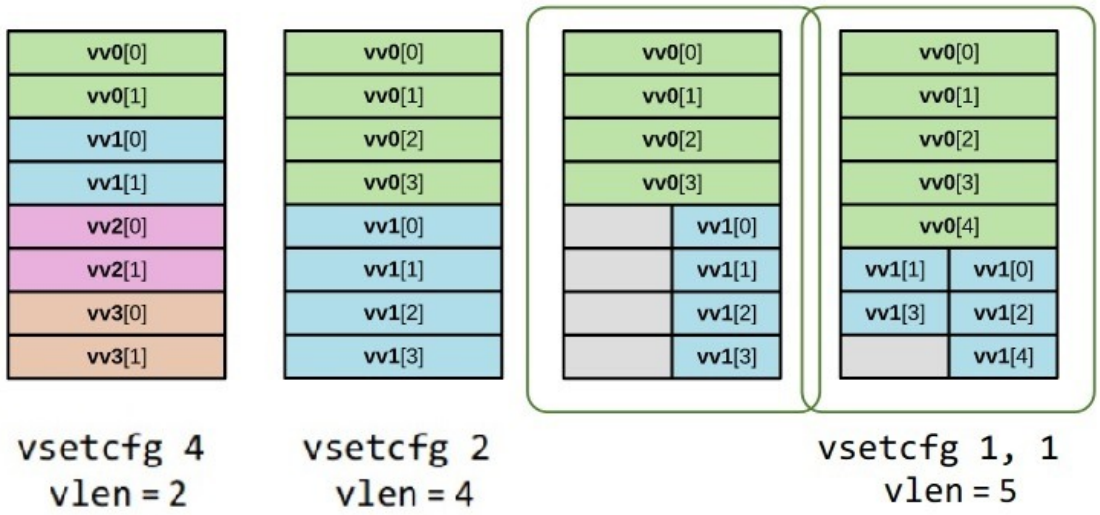


Figura 2.6: Posibles configuraciones de los registros vectoriales de Hwacha [34].

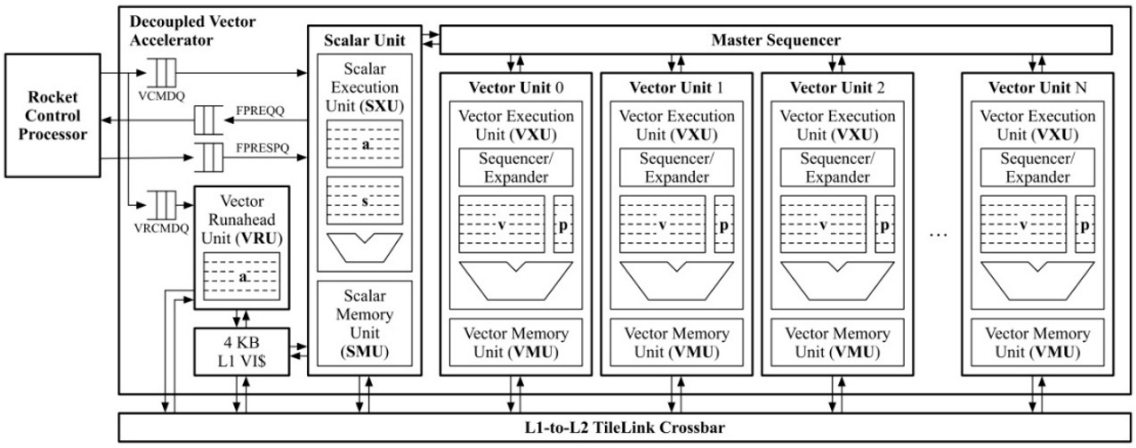


Figura 2.7: Microarquitectura del coprocesador Hwacha [34].

Cabe destacar que los registros vectoriales son configurables vía software. Por ejemplo, como vemos en los primeros elementos de la Figura 2.6, en una aplicación podemos fijar distintos valores para los parámetros *vlen* y *vsetcfg* y conseguir distintas configuraciones de los registros vectoriales. Además, también en esta figura observamos en los últimos dos elementos como Hwacha permite tener una organización con varios tamaños de vector simultáneos.

A continuación, en la Figura 2.7 vemos en detalle su microarquitectura. Observamos que Hwacha se comunica con la unidad de control del procesador para recibir las instrucciones a ejecutar y las guarda en las colas correspondientes. A continuación, las decodifica y las envía a la unidad escalar o a alguna de las unidades vectoriales que lo componen.

Uno de los componentes clave para el rendimiento del coprocesador vectorial Hwacha es, sin duda, los vectores sistólicos que tiene implementados para realizar las multiplicaciones vectoriales directamente con elementos hardware, cuya microarquitectura y localización de los datos por ciclos podemos ver en la Figura 2.8.

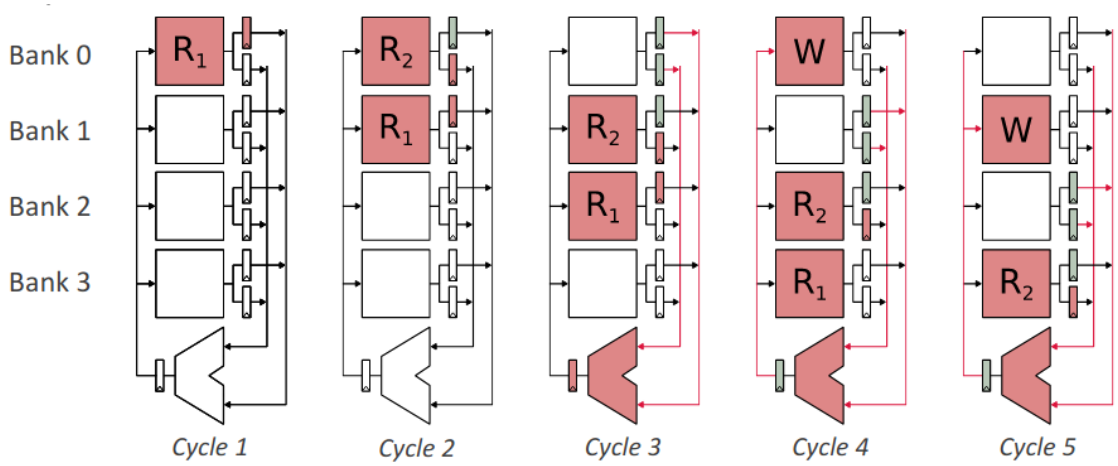


Figura 2.8: Microarquitectura de los vectores sistólicos de Hwacha [34].

2.1.2.2. Gemmini

Gemmini es un acelerador especializado en la multiplicación de matrices integrado en el ecosistema de Chipyard e implementado en Scala, que ejecuta instrucciones personalizadas y no estándar del repertorio RISC-V. En un diseño heterogéneo, el acelerador se comunica con el procesador mediante el puerto RoCC y se conecta directamente a la memoria caché de nivel 2, como podemos ver en la Figura 2.9. Actualmente, es compatible con los cores Rocket y Boom.

El proceso de multiplicación de matrices se lleva a cabo en un vector matricial sistólico para obtener un alto rendimiento. Cada celda del array cuenta con registros entre cada *tile* que la compone, como vemos en la Figura 2.10. La implementación hardware de cada *tile* permite configurar en tiempo de compilación el flujo de las sumas parciales entre *output-stationary* (por filas) o *weight-stationary* (por columnas) [14].

Además, Gemmini cuenta con una memoria *scratchpad* para almacenar los valores intermedios de una multiplicación, una unidad especial para trasponer matrices y un controlador para gestionar la memoria, la comunicación con el core y el tipo de flujo de datos.

2.2. Herramientas

En esta sección vamos a detallar las herramientas que se han utilizado en el desarrollo de este trabajo.

2.2.1. Chipyard

Chipyard [2] es un entorno para el diseño y evaluación de sistemas hardware que se compone de un conjunto de herramientas y librerías diseñadas para proporcionar una vía de integración entre las herramientas *open-source* y las herramientas comerciales para el desarrollo de *sistemas on chip*. El entorno está constituido por los componentes que vemos en la Figura 2.11. Por un lado, tenemos el repertorio de instrucciones RISC-V, el lenguaje FIRRTL para representar los modelos RTL y el lenguaje Chisel, en el que están escritos

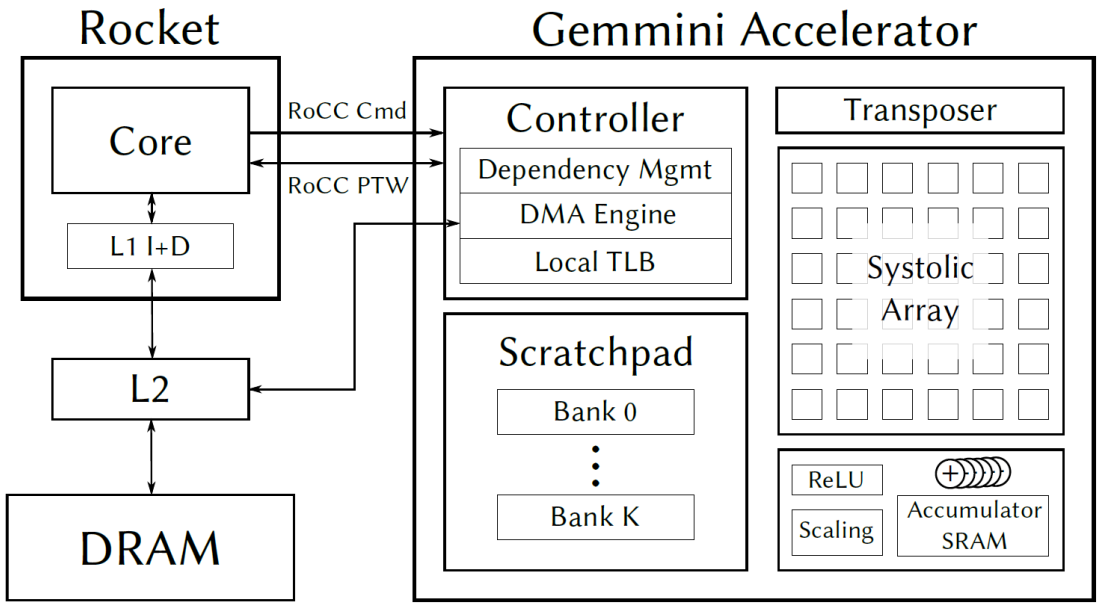


Figura 2.9: Arquitectura del acelerador Gemmini [15].

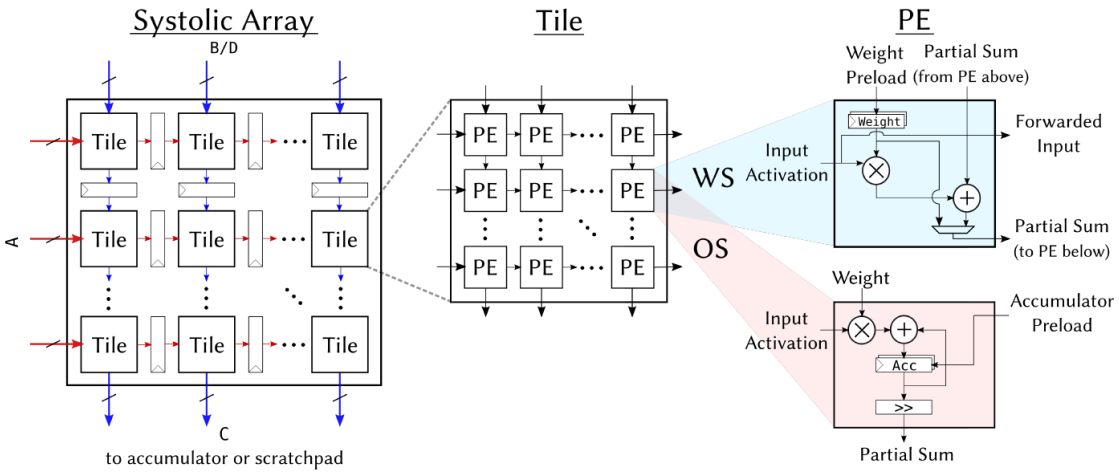


Figura 2.10: Arquitectura del vector sistólico de Gemmini [15].

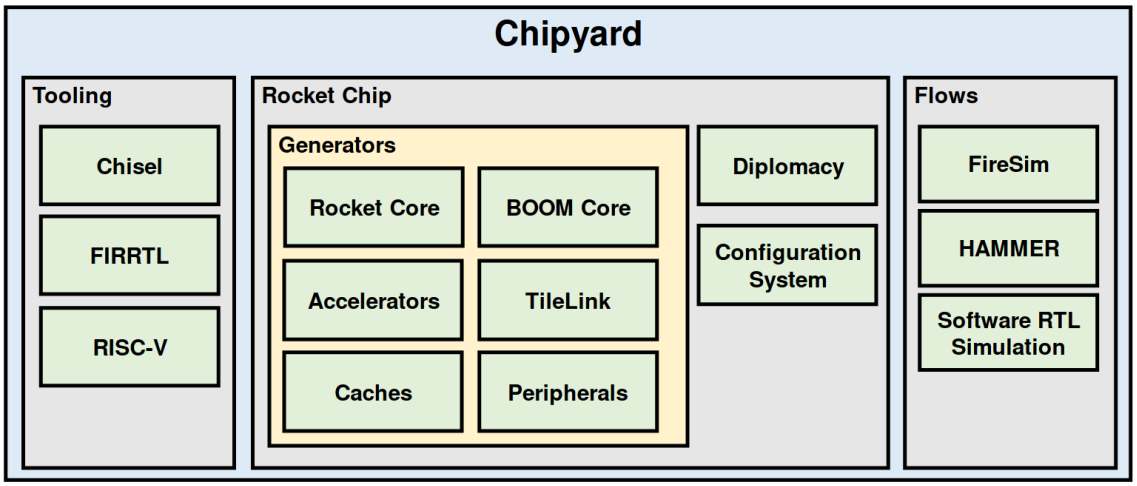


Figura 2.11: Componentes de Chipyard [28].

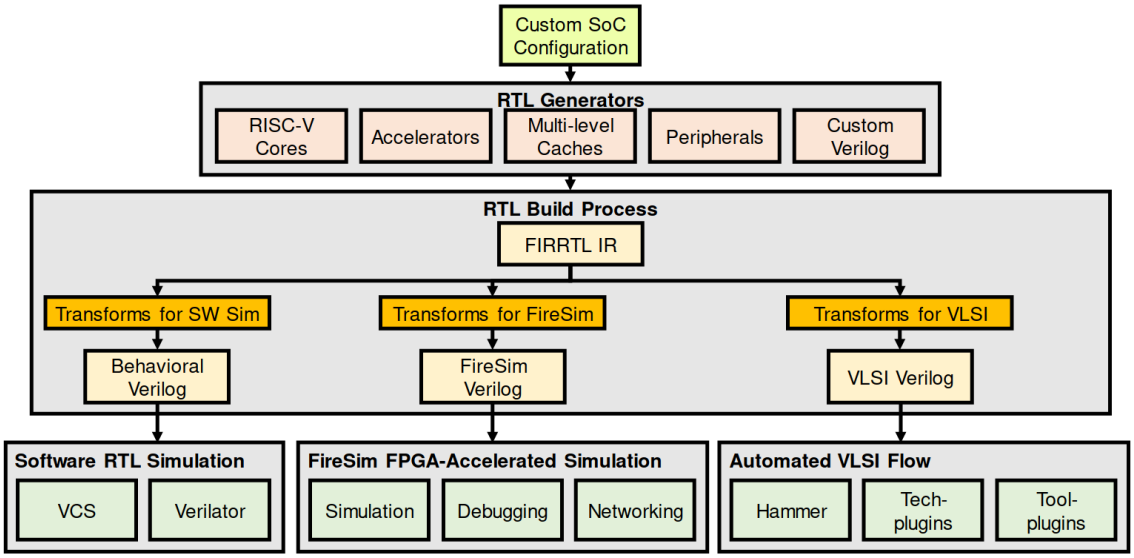


Figura 2.12: Flujo de ejecución en Chipyard [28].

los componentes. En cuanto a los componentes, Chipyard ha embebido y ampliado la herramienta Rocket Chip, ofreciendo algunos cores (Rocket, Boom y CVA6), aceleradores o coprocesadores (Gemmini, Hwacha, SHA3), así como las memorias caché y los sistemas periféricos necesarios para generar un SoC completo. Entre estos componentes se encuentran los cores Rocket y Boom, y los aceleradores Gemmini y Hwacha, que conforman los sistemas heterogéneos escogidos para este trabajo. Finalmente, para llevar estos diseños a un entorno de pruebas, cuenta con herramientas externas como FireSim, para ejecuciones sobre FPGAs en la nube, HAMMER (Higly Agile Masks Made Efforlessly from RTL), para transferir los diseños a VLSI y posteriormente producir el hardware, o la posibilidad de generar Verilog para simular el diseño en herramientas externas como Spike o Qemu.

Además, permite una alta personalización de los diseños hardware generados, ya que gran parte de los componentes de la arquitectura están parametrizados. Por ejemplo, podemos dar el valor deseado a parámetros como el número de entradas del ROB o el número de registros que forman la cola de decodificación.

En el ámbito de simulación de la ejecución de una aplicación sobre un diseño de una arquitectura RISC-V, Chipyard nos ofrece varias posibilidades como vemos en la Figura 2.12. En primer lugar, utilizar un simulador del repertorio de instrucciones RISC-V, como la herramienta Spike, para simular la ejecución de las instrucciones RISC-V que corresponden a la aplicación sin necesitar hardware ni ninguna herramienta adicional. En segundo lugar, podemos integrar Chipyard con FireSim para ejecutar la aplicación en una FPGA en la nube. Finalmente, ofrece la posibilidad de generar el código Verilog para volcarlo sobre una FPGA que implemente RISC-V. La segunda opción es la más versátil y la elegida en este trabajo, ya que permite personalizar el diseño de la arquitectura y ejecutar el código en una FPGA real, encontrando una mayor precisión en la medición de tiempos que en una simulación.

2.2.2. FireSim

FireSim es una plataforma de simulación hardware acelerada por FPGA y con precisión de ciclos [17] enfocada a testear diseños hardware en FPGAs en la nube, en particular, soporta conexión con Amazon Web Services (AWS). FireSim ha sido creado en la Universidad de California en Berkeley y es software libre y abierto.

La plataforma permite tanto ejecutar sobre un nodo en paralelo, como simular centros de datos con distintas topologías de red conectando los nodos. La infraestructura que posee para esto podemos verla en la Figura 2.13. FireSim se instala y se ejecuta sobre una instancia de tipo C4 de AWS llamada *Manager Instance*, ya que desde esta misma se invocan todas las operaciones necesarias para crear y simular un diseño hardware. Sin embargo, esta instancia delegará el trabajo en otras en dos casos: a la hora de crear la imagen necesaria para reproducir un diseño hardware en una FPGA (AFI), y a la hora de simular una ejecución de una aplicación sobre un diseño.

En el primer caso, para cada diseño a generar se crea una instancia del tipo que se haya especificado en cada diseño generando la "Build Farm" que vemos en la Figura 2.13. En el segundo caso, el usuario configura qué tipos de instancias y cuántas de cada tipo se deben crear mediante un fichero de configuración, generando la "Run Farm". Y, posteriormente, se despliega la AFI del diseño hardware elegido en cada una de las instancias y se ejecuta el *workload* correspondiente.

Así, FireSim genera su propio código RTL y ejecuta diversas aplicaciones sobre estas FPGAs en la nube, obteniendo los mismos resultados que si el circuito estuviese implementado físicamente.

2.2.3. FireMarshal

FireMarshal es una herramienta de generación de tareas para sistemas RISC-V expresamente diseñada para integrarse con FireSim [31]. Esta herramienta genera los ficheros de configuración y los ficheros binarios que necesita FireSim para ejecutar una tarea sobre una FPGA. Para ello, necesita el código de una tarea que se quiera ejecutar sobre la FPGA y un fichero de configuración. Algunas de las opciones que se le pueden especificar en el fichero de configuración son las siguientes:

- Base: especificamos la imagen base del sistema de entre tres posibles opciones: bare-

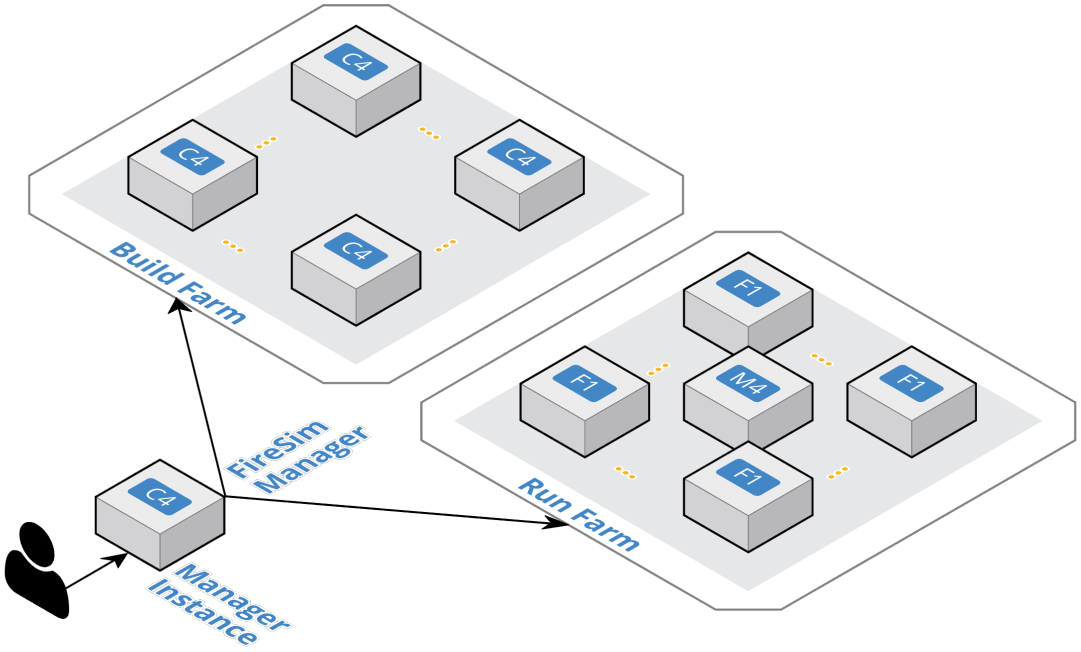


Figura 2.13: Infraestructura de FireSim [18].

metal, fedora o buildroot.

- Host-init: especificamos un script que se ejecutará en el equipo host antes de construir la imagen.
- Run: especificamos un script que se ejecutará en la FPGA una vez configurada.
- Outputs: especificamos una lista de ficheros que se copiarán desde la FPGA al administrador.
- Jobs: especificamos una lista de tareas que se ejecutarán cada una en un nodo en paralelo.

2.2.4. Amazon Web Services

Amazon Web Services [1] es una plataforma de servicios en la nube que ofrece desde cursos de formación en nuevas tecnologías como la inteligencia artificial o el IoT hasta servicios de infraestructura como almacenamiento o cómputo en la nube. En este caso nos centramos en el cómputo en la nube, ya que nos ofrece una gran diversidad de plataformas hardware, entre ellas las instancias EC2 F1, que se corresponden con FPGAs, aportándonos la versatilidad que necesitamos para sintetizar diseños hardware y simular la ejecución de aplicaciones sobre ellos.

Ahora que hemos visto todas las herramientas y los elementos hardware que vamos a utilizar, pasamos a explicar los algoritmos de tratamiento de imágenes que utilizaremos y las modificaciones que les realizamos a los mismos para obtener un mejor rendimiento.

Capítulo 3

Algoritmos de tratamiento de imágenes

En este capítulo, abordamos los algoritmos de tratamiento de imágenes[1] que se van a utilizar como casos de uso, tal como se ha mencionado anteriormente. Más específicamente, se detallarán en este capítulo los algoritmos utilizados para la detección de las líneas de la carretera a partir de las imágenes de la misma. En primer lugar, se presentarán los algoritmos básicos utilizados (algoritmo de Canny y transformada de Hough). A continuación, se mostrará el algoritmo completo para detección de líneas que se ha utilizado como punto de partida en este trabajo y, finalmente, se propondrán modificaciones al mismo que mejoran su eficiencia y rendimiento sin pérdida de precisión.

3.1. Algoritmo de Canny

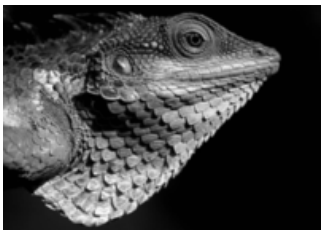
El algoritmo de Canny se encarga de la detección de los bordes en una imagen [7], esta técnica permite extraer información estructural útil de la imagen en cuestión de forma que se reduce la cantidad de datos a procesar en los siguientes pasos del tratamiento. Sin embargo, una técnica de detección de bordes es útil solo si cumple los siguientes requisitos:

- Buena detección - detectar los bordes con una baja tasa de error.
- Buena localización - detectar los puntos de borde de una región con precisión respecto de la imagen real.
- Respuesta mínima - detectar cada borde solo una vez, de forma que se elimine el ruido de los alrededores.

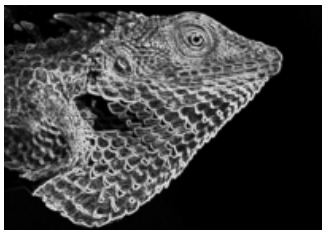
Todos estos requisitos los consigue el algoritmo Canny utilizando el cálculo de variaciones, una técnica matemática que permite encontrar una función analítica que aproxime la curva real con la mayor precisión posible. El procedimiento que sigue el algoritmo de Canny se puede separar en cinco bloques que se detallan a continuación, y se puede ver su efecto sobre una imagen original en la Figura 3.1.

[1] El código correspondiente a los algoritmos de tratamiento de imágenes han sido desarrollados por el profesor Carlos García Sánchez garsanca@dacya.ucm.es de la Facultad de Informática de la UCM.

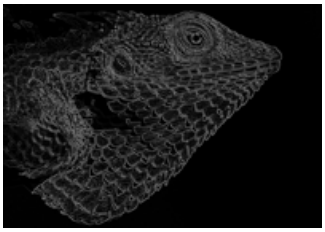
(a) Imagen original.



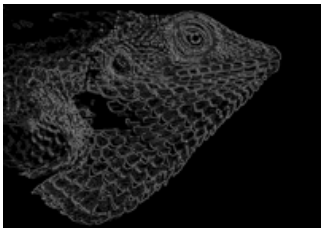
(b) Filtro Gaussiano aplicado.



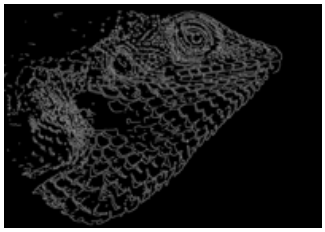
(c) Filtrado por intensidad del gradiente.



(d) Supresión de los no-máximos.



(e) Filtrado por intensidad del gradiente con doble umbral.



(f) Histéresis aplicada.

Figura 3.1: Ejemplo de detección de bordes con el algoritmo de Canny.

1. Reducción de ruido - aplicar el filtro de Gauss para suavizar la imagen.
2. Encontrar el gradiente de intensidad de la imagen.
3. Umbral de magnitud al gradiente - aplicar un umbral al gradiente para descartar los falsos positivos de borde.
4. Doble umbral - aplicar nuevamente un umbral al gradiente para resaltar los bordes potenciales.
5. Histéresis - eliminar los bordes débiles o inconexos.

Entre los métodos de detección de bordes desarrollados hasta ahora, el algoritmo de detección de bordes de Canny es uno de los métodos más estrictamente definidos que proporciona una detección buena y fiable. Debido a su optimización para cumplir los tres criterios de detección de bordes y a la sencillez del proceso de aplicación, se ha convertido en uno de los algoritmos más populares para la detección de bordes.

A continuación, vemos el pseudo-código que hemos utilizado para aplicar el algoritmo de Canny desglosado en cada una de las etapas anteriores. Principalmente se compone de multiplicaciones de matrices consecutivas y chequeo de condiciones para detectar los puntos de borde.

```
void canny(...){
    // Paso 1: Reduccion de ruido
    float NR = mask * image

    //Paso 2: Intensidad del gradiente
    float Gx = mask * NR
    float Gy = mask * NR

    float G = sqrt(Gx^2, Gy^2)
```

```
    float phi = arctan(abs(Gy), abs(Gx))

    //Paso 3: Umbral del gradiente
    if(threshold(phi[*])){
        float phi[*] in {0, 45, 90, 135}
    }

    //Paso 4: Doble umbral
    if (condition(phi[*]) && condition(G)){
        int edge[*] = 1;
    }

    //Paso 5: Histeresis
    if (condition(G[*]) && edge[*]){
        int image_out[*] = 255;
    }
}
```

3.2. Transformada de Hough

La transformada de Hough es una técnica de extracción de características que se utiliza en numerosos campos que requieren el procesamiento de imágenes como la visión por ordenador o el procesamiento digital de imágenes. El objetivo del algoritmo es encontrar objetos imperfectos de entre unas determinadas clases de objetos mediante un procedimiento de votación. Este procedimiento consiste en crear un espacio con los valores asignados a cada píxel, de forma que los máximos locales resultantes en el espacio acumulador son los posibles objetos detectados.

La transformada de Hough clásica tan solo se aplicaba para la detección de líneas rectas, sin embargo, se ha ido modificando y actualmente se utiliza para la detección de curvas arbitrarias como elipses o círculos.

A continuación, vemos el código que hemos utilizado para aplicar la transformada de Hough. En este código para cada punto de borde que se ha detectado previamente con el algoritmo de Canny, traza un haz de líneas que pasan por él y guarda la cantidad de líneas que pasan por cada píxel de la imagen. De esta forma, los puntos por donde más líneas pasen se corresponderán con una línea en la imagen original.

```
void houghtransform(...){
    //Para cada punto de borde
    if( image[ (i*width) + j] > 250 ){
      for(theta=0;theta<180;theta++) //Haz de lineas
      {
        float rho = ( center(j) * cos(theta) + center(i) * sin(theta);
        accumulators[ (rho + cte) * 180 + theta]++;
      }
    }
}
```

3.3. Detección de líneas

Una vez vistos los algoritmos anteriores, vamos a utilizar una combinación de ambos junto con otro código específico para detectar con mayor precisión las líneas que delimitan los carriles en las carreteras convencionales.

Para ello, dada una imagen de entrada, aplicamos en primer lugar el algoritmo de Canny y, a continuación, la transformada de Hough, de forma que este último algoritmo nos devolverá una matriz de las mismas dimensiones de la imagen con unos valores de acumulación. Las celdas que se correspondan con un máximo local dentro de la matriz nos darán las coordenadas polares de una recta de la imagen original. A continuación, vemos el pseudo-código de dicha función que consiste en buscar los máximos locales de la imagen preprocesada y extraer las coordenadas de la recta que representa.

```
void getlines(...){
    //Para cada punto de la imagen (rho, theta)
    //Si es un maximo local
    if(accumulators[*] >= threshold){
      max = accumulators[*];
      //Revisamos en su vecindad
      if( accumulators[vecindad(*))] > max ){
        max = accumulators[vecindad(*)];
      }
    }
    //Guardamos dos puntos que delimitan la recta
    lines.add(x1, y1, x2, y2);
}
```

3.4. Modificación de los tipos de datos

En los apartados anteriores, hemos presentado el código original de los algoritmos. Sin embargo, este código utiliza muchas variables en punto flotante cuyas operaciones son computacionalmente más costosas que las operaciones sobre enteros. Por tanto, es conveniente sustituir las variables de punto flotante por variables enteras siempre que no se pierda precisión.

Primero, notamos que el algoritmo de Canny es dónde se usan más variables en punto flotante que se pueden sustituir por enteras, de forma que utilizamos menos espacio en memoria. Además, hay más recursos de hardware empotrado preparados para operaciones con números enteros cortos que con punto flotante, por ejemplo [4] [22] [21]. Veamos a continuación las variables que se pueden sustituir sin la correspondiente pérdida de precisión.

- **NR:** Esta variable es una matriz dónde se guarda la imagen resultante al aplicar una máscara dada por la matriz

$$\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix}$$

Figura 3.2: Imagen de salida con las líneas detectadas en rojo.

a las vecindades 5x5 de cada píxel de la imagen original. Dado que la máscara contiene números enteros y la imagen original viene dada por enteros, podemos declarar la variable como entera sin pérdida de precisión.

- **Gx** y **Gy:** Estas variables son matrices donde se guardan las trasformaciones dadas por las siguientes matrices

$$\begin{pmatrix} 1 & 2 & -2 & -1 \\ 4 & 8 & -8 & -4 \\ 6 & 12 & -12 & -6 \\ 4 & 8 & -8 & -4 \\ 1 & 2 & -2 & -1 \end{pmatrix}, \quad \begin{pmatrix} -1 & -2 & 2 & 1 \\ -4 & -8 & 8 & 4 \\ -6 & -12 & 12 & 6 \\ -4 & -8 & 8 & 4 \\ -1 & -2 & 2 & 1 \end{pmatrix}$$

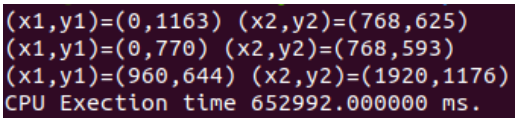
aplicadas a las vecindades 5x4 de cada elemento de la variable NR anterior. Dado que NR la hemos transformado en entera y las máscaras contienen números enteros con signo, podemos declarar las variables Gx y Gy como enteras sin pérdida de precisión.

- **Phi:** En el código original la variable phi recibe un valor flotante correspondiente al arctan $\frac{Gy}{Gx}$, que es un número flotante. Sin embargo, esta asignación es temporal y a continuación se le da un valor entero en $\{0, 45, 90, 135\}$ a cada elemento de la matriz phi. Por tanto, para optimizar el código, guardaremos los valores temporales flotantes en una variable auxiliar que posteriormente se eliminará, y se declarará la variable phi como entera.

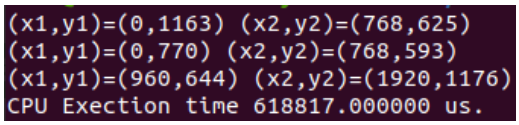
Una vez realizados estos cambios en el código, comprobamos que no hemos perdido precisión en la detección de líneas en una imagen. En la Figura 3.2 vemos la imagen original con las líneas detectadas marcadas en rojo y en la Figura 3.3a vemos el resultado analítico correspondiente a dichas líneas. Comprobamos en la Figura 3.3b que el resultado de la ejecución del código modificado se corresponde con la ejecución del código anterior, y además ha sido ligeramente más rápida.

3.5. Profiling del código

A continuación, vamos a realizar un *profiling* del código para determinar en qué partes se emplea un mayor número de recursos y tiempo. Para ello, medimos el tiempo que se



(a) Ejecución del código original.



(b) Ejecución del código modificado.

Figura 3.3: Resultados de las ejecuciones de los distintos códigos.

	Etapa 1	Etapa 2	Etapa 3	Total
Tiempo (μs)	43803	98171	456030	598004
Porcentaje respecto el total	7,32 %	16,42 %	76,26 %	

Tabla 3.1: *Profiling* del código completo por etapas.

emplea en cada uno de los algoritmos anteriores y lo comparamos respecto al tiempo total de la ejecución. Dado que el tiempo de ejecución depende de variables de entorno del sistema donde se ejecuta, los tiempos difieren entre ejecuciones. Para solucionarlo se toma la media de los tiempos de tres ejecuciones consecutivas.

El código a ejecutar se puede dividir en las siguientes tres etapas: carga de la imagen, donde se lee la imagen desde un fichero y se le aplica un filtro en blanco y negro utilizando la librería externa "libpng"; tratamiento de la imagen, donde se le aplican los algoritmos vistos anteriormente en esta sección; y, generación de la salida, donde se modifica la imagen inicial para añadirle las líneas encontradas.

En un primer *profiling* cuyos resultados vemos en la Tabla 3.1, apreciamos que la etapa 3 tiene el mayor consumo de tiempo, un $76,26\,\%$ del total. Esta etapa es la encargada de generar la imagen de salida con las líneas impresas en ella, sin embargo, en un entorno IoT, no necesitamos un resultado visual, nos basta con las coordenadas de las líneas. Por tanto, modificamos el código eliminando la creación de la imagen de salida.

De esta forma, la etapa 3 consume alrededor de $4\mu s$, como vemos en la Tabla 3.2, resultando irrelevante su papel en el rendimiento del programa. Ahora, abordamos la etapa 2, ya que consume casi el $70\,\%$ del tiempo de ejecución del nuevo programa sin salida gráfica. Esta etapa a su vez se subdivide en las siguientes tres fases: algoritmo de Canny, transformada de Hough y detección de líneas. Así pues, realizamos un *profiling* de la etapa 2 subdividida, que podemos ver en la Tabla 3.3. Con esto, nos damos cuenta de que la fase 1 que se corresponde con la aplicación del algoritmo de Canny supone la mayor parte del tiempo, un $87,64\,\%$.

Dado que el algoritmo de Canny se basa en la multiplicación de matrices, debemos diseñar un hardware específico para acelerar esta operación tan costosa computacionalmente.

	Etapa 1	Etapa 2	Etapa 3	Total
Tiempo (μs)	43485	98714	4	142203
Porcentaje respecto el total	30,58 %	69,42 %	0,00 %	

Tabla 3.2: *Profiling* del código sin imagen de salida por etapas.

	Fase 1	Fase 2	Fase 3	Total
Tiempo (μs)	90265	12275	459	102999
Porcentaje respecto el total	87,64 %	11,92 %	0,45 %	

Tabla 3.3: *Profiling* de la etapa 2 por fases.

3.6. Vectorización del código

En esta sección vamos a vectorizar el código lo máximo posible para poder realizar las operaciones de multiplicación de vectores de forma más eficiente. En particular, se van a desarrollar y vectorizar las operaciones del algoritmo de Canny.

Este algoritmo, en su primera fase, calcula una matriz de forma que cada celda (i, j) es el resultado de la suma de la multiplicación elemento a elemento de su vecindad 5x5 por una máscara 5x5 definida previamente dividido por la constante 159. En el código original estas operaciones se realizan con dos bucles anidados que recorren la matriz original como podemos ver a continuación.

```
for(i=2; i<height-2; i++)
    for(j=2; j<width-2; j++)
    {
      // Noise reduction
      NR[i*width+j] =
         (2*im[(i-2)*width+(j-2)] +  4*im[(i-2)*width+(j-1)] +  5*im[(i-2)*width+(j)] +  4*im[(i-2)*width+(j+1)] + 2*im[(i-2)*width+(j+2)]
        + 4*im[(i-1)*width+(j-2)] +  9*im[(i-1)*width+(j-1)] + 12*im[(i-1)*width+(j)] +  9*im[(i-1)*width+(j+1)] + 4*im[(i-1)*width+(j+2)]
        + 5*im[(i  )*width+(j-2)] + 12*im[(i  )*width+(j-1)] + 15*im[(i  )*width+(j)] + 12*im[(i  )*width+(j+1)] + 5*im[(i  )*width+(j+2)]
        + 4*im[(i+1)*width+(j-2)] +  9*im[(i+1)*width+(j-1)] + 12*im[(i+1)*width+(j)] +  9*im[(i+1)*width+(j+1)] + 4*im[(i+1)*width+(j+2)]
        + 2*im[(i+2)*width+(j-2)] +  4*im[(i+2)*width+(j-1)] +  5*im[(i+2)*width+(j)] +  4*im[(i+2)*width+(j+1)] + 2*im[(i+2)*width+(j+2)])
        /159;
    }
```

Sin embargo, para poder multiplicar vectores de un tamaño considerable hemos modificado el código como sigue: en primer lugar, se prepara un vector con la máscara replicada N veces y otro vector con las vecindades correspondientes a N celdas (i, j); a continuación, se multiplican dichos vectores mediante un hardware de propósito específico; y, finalmente, se suman los elementos del vector resultante de la multiplicación en grupos de 25 elementos, ya que el valor de cada celda (i, j) se corresponde con la multiplicación de su vecindad 5x5 y su máscara 5x5, y se divide por 159 para darle valor a la celda. Así, se calculan los valores de N celdas de la matriz con una operación de multiplicación de vectores. A continuación vemos el código resultante.

```
for(i=2; i<height-2; i++)
    for(j=2; j<width-2; j++)
    {
      // Noise reduction
      NR[i*width+j] =
         (2*im[(i-2)*width+(j-2)] +  4*im[(i-2)*width+(j-1)] +  5*im[(i-2)*width+(j)] +  4*im[(i-2)*width+(j+1)] + 2*im[(i-2)*width+(j+2)]
```

```
7         + 4*im[(i-1)*width+(j-2)] +  9*im[(i-1)*width+(j-1)] + 12*im[(i-1)*
      width+(j)] +  9*im[(i-1)*width+(j+1)] + 4*im[(i-1)*width+(j+2)]
8         + 5*im[(i  )*width+(j-2)] + 12*im[(i  )*width+(j-1)] + 15*im[(i  )*
      width+(j)] + 12*im[(i  )*width+(j+1)] + 5*im[(i  )*width+(j+2)]
9         + 4*im[(i+1)*width+(j-2)] +  9*im[(i+1)*width+(j-1)] + 12*im[(i+1)*
      width+(j)] +  9*im[(i+1)*width+(j+1)] + 4*im[(i+1)*width+(j+2)]
10        + 2*im[(i+2)*width+(j-2)] +  4*im[(i+2)*width+(j-1)] +  5*im[(i+2)*
      width+(j)] +  4*im[(i+2)*width+(j+1)] + 2*im[(i+2)*width+(j+2)])
11        /159;
12    }
```

El algoritmo de Canny contiene otra fase donde se realiza el mismo procedimiento para calcular nuevas matrices con valores auxiliares que se utilizan en el resultado final del tratamiento de la imagen. Luego, análogamente, se modifican sus bucles de multiplicaciones de elementos.

Ahora, estamos en disposición de pasar al siguiente capítulo, donde se generan distintas arquitecturas sobre las que ejecutar los algoritmos vistos de forma eficiente y se discuten los resultados obtenidos.

Capítulo 4

Resultados

En este capítulo, vamos a presentar los resultados que hemos obtenido con las herramientas y los elementos hardware que hemos presentado anteriormente. En primer lugar, veremos cómo se generan las imágenes de los diseños de las arquitecturas y los diseños particulares que se han generado. En segundo lugar, veremos cómo se generan los *workload* que se ejecutarán sobre esos diseños. En tercer lugar, vemos el proceso a seguir para realizar una simulación de la ejecución de un *workload* sobre una arquitectura determinada. Finalmente, veremos los experimentos que se han realizado con distintos *workload* y sobre distintas arquitecturas y los resultados de rendimiento que se han obtenido. El código uitlizado para la generación de los diseños, así como el código de los *workload* y su correspondiente configuración pueden encontrarse en `https://github.com/mbelda/Autonomous-vehicles-tests`.

4.1. Generación de diseños de arquitecturas

En esta sección, detallamos los pasos a seguir para generar una imagen FPGA (AGFI) de un diseño de una arquitectura. Esta imagen será usada posteriormente para configurar una FPGA cloud de AWS que simule la arquitectura diseñada.

Estas arquitecturas se definen por partes utilizando las clases de FireSim y Chipyard. El primer paso es generar una configuración con el tipo y número de procesadores, los coprocesadores o aceleradores y la base de la arquitectura. Esta definición se encuentra en un módulo de Chipyard específico para ello en la ruta `firesim/target-design/chipyard/generators/chipyard/src/main/scala/config`. En esta ruta encontramos un fichero con nomenclatura *[nombreProcesador]Configs.scala* para cada procesador donde se definen sus arquitecturas. Por ejemplo, para nuestra arquitectura heterogénea unicore con un Rocket y el coprocesador vectorial Hwacha tenemos la siguiente configuración en el fichero *RocketConfigs.scala*, donde vemos como añadimos cada módulo de menor a mayor relevancia: la configuración base, el procesador Rocket y el coprocesador vectorial Hwacha. Este orden es importante, ya que las configuraciones se sobreescriben las unas a las otras y predomina la que se encuentre más arriba.

```
1 class HwachaRocketConfig extends Config(
2   new chipyard.config.WithHwachaTest ++
3   new hwacha.DefaultHwachaConfig ++
```

```
4   new freechips.rocketchip.subsystem.WithNBigCores(1) ++
5   new chipyard.config.AbstractConfig)
```

A continuación, debemos definir la arquitectura completa en el módulo de FireSim correspondiente haciendo uso de la anterior configuración. En la ruta `firesim/deploy` encontramos todos los ficheros de configuración necesarios para una ejecución completa de un *workload* en una determinada arquitectura. En particular, para indicar la configuración de la arquitectura utilizamos el fichero *config_ build_ recipes.ini* donde añadimos una entrada con el formato que vemos a continuación para cada arquitectura.

```
1 [firesim-rocket-singlecore-hwacha-no-nic-l2-llc4mb-ddr3-MCRams-50MHz]
2 DESIGN=FireSim
3 TARGET_CONFIG=
      DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_WithFireSimTestChipConfig
4     Tweaks_chipyard.HwachaRocketConfig
5 PLATFORM_CONFIG=MCRams_F50MHz_BaseF1Config
6 instancetype=z1d.2xlarge
7 deploytriplet=None
```

En primer lugar, tenemos entre corchetes el nombre de la arquitectura, que deberá ser único. En segundo lugar, tenemos la variable *DESIGN* donde indicamos que se encargue de generar el diseño la herramienta FireSim. A continuación, contamos con las variables *TARGET_ CONFIG* y *PLATFORM_ CONFIG* donde indicamos las características específicas de la propia arquitectura. Presentamos el significado de las opciones relevantes que utilizamos:

- DDR3FRFCFSLLC4MB: Añade una memoria RAM L2 de 16GB y una L1 de 4MB asociativa con 8 vías.
- WithFireSimTestChipConfigTweaks: Añade un reloj con una frecuencia de 1 GHz para comunicarse con la memoria RAM que viene configurada con la misma frecuencia de reloj.
- WithDefaultFireSimBridges: Añade los circuitos hardware necesarios para establecer la comunicación con dispositivos como el puerto serie o el puerto UART.
- MCRams: Habilita un mecanismo para sustituir las memorias RAM multipuerto, tipicamente ineficientes, por un modelo desacoplado que simula la memoria RAM mediante accesos serializados a una implementación subyacente que se puede mapear a una RAM de bloque (BRAM) eficiente [18].
- FXMHz: Frecuencia de reloj del procesador fijada a X MHz. Debido a que algunos diseños heterogéneos son complejos a nivel de circuitos necesitamos elegir un valor de frecuencia bajo para que se pueda sintetizar el diseño. Así pues, elegimos 80MHz para la mayoría de las arquitecturas (veremos más adelante alguna excepción impuesta por limitaciones tecnológicas) con el fin de compararlas entre sí de la forma más justa posible.
- MTModels: Habilita una optimización que se encarga de separar los nodos y ejecutarlos en un hilo de ejecución independiente en una implementación física subyacente compartida. Obteniendo así los mismos resultados, pues se conserva la precisión ciclo

a ciclo y bit a bit, y se pueden generar diseños más grandes con mayor número de cores.

Finalmente, las variables *instancetype* y *deploytriplet* sirven para indicar configuraciones de la instancia de AWS. Simplemente destacamos que en la primera de ellas se indica el tipo de la instancia a utilizar para dicha arquitectura.

Llegados a este punto, contamos con la configuración completa de un diseño de una arquitectura. Ahora bien, para generarla debemos indicarlo en uno de los ficheros de configuración de FireSim. En particular, el fichero *config_build.ini* contiene los nombres de las arquitecturas que se quieren generar. Por tanto, el procedimiento a seguir sería añadir el nombre de las arquitecturas que queremos generar por filas en dicho fichero y eliminar el resto de arquitecturas que pueda haber, pues en caso de no eliminar las restantes se volverían a generar y sobreescribirían la imagen anterior.

Para terminar, desde el propio directorio `firesim/deploy` en el que se encuentra tanto la herramienta *firesim* como los ficheros de configuración, ejecutamos el comando *firesim buildafi* para que se generen las AGFI de las arquitecturas que se han definido. Una vez finalizada la ejecución de este comando, obtenemos unos parámetros que se corresponden con el nombre de la imagen y sus características, similar al que vemos a continuación.

```
1 [firesim-rocket-singlecore-hwacha-no-nic-l2-llc4mb-ddr3-MCRams-80MHz-noILA]
2 agfi=agfi-0a08242b1ddef3f81
3 deploytripletoverride=None
4 customruntimeconfig=None
```

Para que la herramienta FireSim encuentre la imagen que se ha generado y se pueda utilizar en futuras ejecuciones, debemos añadir estas líneas al fichero *config_hwdb.ini* en el propio directorio `firesim/deploy`. Este fichero, por tanto, debe contener la información de cada una de las imágenes generadas.

Una vez visto el proceso que hay que seguir para generar una imagen de un diseño de una arquitectura y el proceso de configuración que se debe llevar a cabo para poder utilizar la arquitectura en las simulaciones, pasamos a ver en detalle las arquitecturas que hemos generado.

4.2. Arquitecturas generadas

A continuación presentamos los diseños de las arquitecturas que se han utilizado en el desarrollo de este trabajo. Los componentes principales que se han utilizado para generar las arquitecturas son los procesadores Rocket y Boom, el acelerador de multiplicación de matrices Gemmini y el coprocesador vectorial Hwacha.

El resto de componentes de una arquitectura como la memoria, la frecuencia de reloj, los buses, etc serán los mismos para las distintas arquitecturas con el fin de compararlas de la forma más justa posible. Todas las arquitecturas utilizan las opciones DDR3FRFCFSLLC4MB, WithDefaultFireSimBridges y WithFireSimTestChipConfigTweaks en el campo *TARGET_CONFIG* y las opciones MCRams y BaseF1Config en el campo *PLATFORM_CONFIG*. Además, las arquitecturas homogéneas multicore incluyen también en este segundo campo la opción MTModels.

En cuanto a la frecuencia de reloj de la arquitectura, que se define mediante la opción FXMHz en el campo *TARGET_ CONFIG*, hemos generado arquitecturas a dos frecuencias distintas, a 80 y 50MHz. Esto es debido a que las arquitecturas que utilizan el acelerador Gemmini son más complejas a nivel de circuito y requieren 50MHz de frecuencia o menos para poder sintetizar el diseño. Sin embargo, el resto de arquitecturas heterogéneas y multicore se han podido generar a 80MHz. Por tanto, se han generado también las arquitecturas básicas homogéneas multicore a 50MHz, con el fin de comparar justamente los tiempos de las ejecuciones utilizando el acelerador Gemmini.

Una vez vistas las opciones generales del diseño, pasamos a ver en detalle las características de los procesadores, los aceleradores y los coprocesadores. Todos los componentes utilizados están escritos en Scala, lo que nos ofrece cierta facilidad para modificar propiedades básicas de los elementos hardware como el número de registros de cada tipo, el número de entradas en el ROB, etc. Las configuraciones particulares elegidas para estos valores se representan mediante una clase con nomenclatura */CaracteristicasDeLaArquitectura/Config*, que extiende a su vez la clase genérica Config. Y es esta clase, que determina los detalles de la arquitectura, la que se utiliza en el campo *TARGET_ CONFIG*, como hemos visto anteriormente. Por tanto, pasemos a ver en detalle las clases que se han utilizado en cada arquitectura.

1. Arquitectura homogénea unicore con un procesador Rocket.
   La configuración de esta arquitectura viene dada por la siguiente clase.

```
1    class RocketConfig extends Config(
2    new freechips.rocketchip.subsystem.WithNBigCores(1) ++
3    new chipyard.config.AbstractConfig)
```

   El fragmento *WithNBigCores* es la parte relevante de esta configuración, pues define las características del procesador. En este caso vemos que se hace referencia a un core *Big*, sin embargo existen macros análogas para los tamaños *Medium, Small* y *Tiny* del procesador Rocket. La principal diferencia entre el resto de tamaños y el *Big* que utilizamos es que los restantes no cuentan con unidad de punto flotante (FPU) ni predictor de saltos (BTB). Por ejemplo, mostramos la configuración de *WithNSmallCores* dónde podemos observar como se indica en las líneas 6 y 7 que no contiene FPU ni BTB.

```
1    class WithNSmallCores(n: Int, overrideIdOffset: Option[Int] = None
     ) extends Config((site, here, up) => {
2      case RocketTilesKey => {
3        val prev = up(RocketTilesKey, site)
4        val idOffset = overrideIdOffset.getOrElse(prev.size)
5        val small = RocketTileParams(
6          core = RocketCoreParams(useVM = false, fpu = None),
7          btb = None,
8          dcache = Some(DCacheParams(
9            rowBits = site(SystemBusKey).beatBits,
10           nSets = 64,
11           nWays = 1,
12           nTLBSets = 1,
13           nTLBWays = 4,
14           nMSHRs = 0,
15           blockBytes = site(CacheBlockBytes))),
```

```
16          icache = Some(ICacheParams(
17            rowBits = site(SystemBusKey).beatBits,
18            nSets = 64,
19            nWays = 1,
20            nTLBSets = 1,
21            nTLBWays = 4,
22            blockBytes = site(CacheBlockBytes))))
23        List.tabulate(n)(i => small.copy(hartId = i + idOffset)) ++
    prev
24      }
25    })
```

En cuanto al resto de configuración, la clase *AbstractConfig* se encarga de dar valores al resto de elementos de una arquitectura necesarios para su funcionamiento, como los buses, los elementos de entrada/salida, etc. Es por ello que esta clase estará presente en todas las configuraciones que generemos.

2. Arquitectura homogénea multicore con dos procesadores Rocket.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
1   class DualRocketConfig extends Config(
2     new freechips.rocketchip.subsystem.WithNBigCores(2) ++
3     new chipyard.config.AbstractConfig)
```

   Observamos que la definición de la clase es análoga a la anterior con la excepción de que el parámetro de número de cores que le pasamos a la clase *WithNBigCores* tiene el valor 2. Por tanto, generaremos una arquitectura con dos cores Rocket con las mismas características que el procesador de la arquitectura unicore, con el fin de poder comparar los resultados de las ejecuciones de las mismas aplicaciones sobre ambas.

3. Arquitectura heterogénea unicore con un procesador Rocket y el coprocesador vectorial Hwacha.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
1   class HwachaRocketConfig extends Config(
2     new chipyard.config.WithHwachaTest ++
3     new hwacha.DefaultHwachaConfig ++
4     new freechips.rocketchip.subsystem.WithNBigCores(1) ++
5     new chipyard.config.AbstractConfig)
```

   En esta arquitectura, observamos que además del core Rocket de tamaño *Big* añadimos el coprocesador vectorial Hwacha mediante la clase *DefaultHwachaConfig* y unos tests propios de Hwacha para comprobar su correcto funcionamiento al generar el diseño mediante la clase *WithHwachaTest*.

   En el caso de Hwacha, la configuración también es fácilmente modificable como en los procesadores, ya que viene escrita en Scala y están todas las características parametrizadas. A continuación presentamos la configuración por defecto que utilizamos para Hwacha, donde vemos entre otras características que la memoria caché es directa y tiene 64 conjuntos, contamos con 256 registros vectoriales y 64 escalares y el tamaño máximo de vector es de 256 bytes.

```
class DefaultHwachaConfig extends Config((site, here, up) => {
    case HwachaIcacheKey => ICacheParams(
      nSets = 64,
      nWays = 1,
      rowBits = 1 * 64,
      nTLBWays = 8,
      fetchBytes = 8,
      latency = 1
    )

    case HwachaCommitLog => true

    // hwacha constants
    case HwachaNAddressRegs => 32
    case HwachaNScalarRegs => 64
    case HwachaNVectorRegs => 256
    case HwachaNPredRegs => 16
    case HwachaRegBits => math.max(log2Up(site(HwachaNVectorRegs))
, log2Up(site(HwachaNScalarRegs)))
    case HwachaPredRegBits => log2Up(site(HwachaNPredRegs))
    case HwachaRegLen => 64
    case HwachaMaxVLen =>
      site(HwachaNBanks) * site(HwachaNSRAMRFEntries) *
        site(HwachaBankWidth) / site(HwachaRegLen)

    case HwachaNDTLB => 8
    case HwachaNPTLB => 4
    case HwachaLocalScalarFPU => false

    // Multi-lane constants
    case HwachaNLanes => 1

    // lane constants
    case HwachaBankWidth => 128
    case HwachaNBanks => 4
    case HwachaNSRAMRFEntries => 256
    case HwachaNFFRFEntries => 16
    case HwachaNFFRFReadPorts => 3
    case HwachaNPredRFEntries => 256
    case HwachaNPredRFReadPorts => 3
    case HwachaNOperandLatches => 6
    case HwachaNPredLatches => 4
    case HwachaWriteSelects => 2
    case HwachaRFAddrBits => math.max(log2Up(site(
HwachaNSRAMRFEntries)), log2Up(site(HwachaNFFRFEntries)))
    case HwachaPRFAddrBits => log2Up(site(HwachaNPredRFEntries))

    case HwachaStagesALU => 1
    case HwachaStagesPLU => 0
    case HwachaStagesIMul => 3
    case HwachaStagesDFMA => 4
    case HwachaStagesSFMA => 3
    case HwachaStagesHFMA => 3
    case HwachaStagesFConv => 2
    case HwachaStagesFCmp => 1

    case HwachaNSeqEntries => 8

    case HwachaNVVAQEntries => 4
    case HwachaNVPAQEntries => 24
    case HwachaNVSDQEntries => 4
```

```
        case HwachaNVLDQEntries => 4
        case HwachaNVMTEntries => 64

        case HwachaNSMUEntries => 16
        case HwachaBuildVRU => true

        case BuildRoCC => up(BuildRoCC) ++ Seq(
          (p: Parameters) => {
             val hwacha = LazyModule.apply(new Hwacha()(p))
             hwacha
          }
        )
        // Set TL network to 128bits wide
        case SystemBusKey => up(SystemBusKey, site).copy(beatBytes =
    16)

        case HwachaConfPrec => false
        case HwachaVRUMaxOutstandingPrefetches => 20
        case HwachaVRUEarlyIgnore => 1
        case HwachaVRUMaxRunaheadBytes => 16777216
        case HwachaCMDQLen => 32
        case HwachaVSETVLCompress => true
      }
    )
```

4. Arquitectura heterogénea unicore con un procesador Rocket y el acelerador de multiplicación de matrices Gemmini.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
    class GemminiRocketConfig extends Config(
      new gemmini.DefaultGemminiConfig ++
      new freechips.rocketchip.subsystem.WithNBigCores(1) ++
      new chipyard.config.AbstractConfig)
```

   En esta arquitectura, observamos que además del core Rocket de tamaño *Big* añadimos el acelerador Gemmini mediante la clase *DefaultGemminiConfig*. En el caso de Gemmini, la configuración por defecto es la siguiente.

```
    class DefaultGemminiConfig[T <: Data : Arithmetic, U <: Data, V <:
     Data](
      gemminiConfig: GemminiArrayConfig[T,U,V] = GemminiConfigs.
    defaultConfig
    ) extends Config((site, here, up) => {
      case BuildRoCC => up(BuildRoCC) ++ Seq(
        (p: Parameters) => {
          implicit val q = p
          val gemmini = LazyModule(new Gemmini(gemminiConfig))
          gemmini
        }
      )
      case SystemBusKey => up(SystemBusKey).copy(beatBytes = 16)
    })
```

5. Arquitectura homogénea unicore con un procesador Boom.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
1    class LargeBoomConfig extends Config(
2    new boom.common.WithNLargeBooms(1) ++
3    new chipyard.config.AbstractConfig)
```

El fragmento *WithNLargeBooms* es la parte relevante de esta configuración, pues define las características del procesador. En este caso vemos que se hace referencia a un core *Large*, sin embargo existen macros análogas para los tamaños *Giga, Mega, Large, Medium* y *Small* del procesador Boom. La principal diferencia entre el resto de tamaños y el *Large* que utilizamos es el tamaño de los componentes internos como la memoria L1 o el ROB (re-order buffer). Por ejemplo, mostramos la configuración de *WithNLargeBooms* dónde podemos observar en la línea 13 que el procesador tiene 96 entradas en el ROB y en la línea 28 vemos que la configuración del número de vías del TLB (translation lookaside buffer) de la memoria caché L1 de datos es de 16. Sin embargo, si consultamos la configuración de la clase *WithNSmallBooms* y *WithNMegaBooms* podemos ver que el número de entradas del ROB es 32 y 128, respectivamente, y la configuración del número de vías del TLB de la memoria caché L1 de datos es 8 y 32, respectivamente. Se ha elegido el core de tamaño **Large** para que sea suficientemente grande para proporcionarnos buen rendimiento sin excederse, ya que estamos en un entorno IoT donde es importa no escedernos en tamaño ni consumo.

```
1    class WithNLargeBooms(n: Int = 1, overrideIdOffset: Option[Int] =
     None) extends Config(
2      new WithTAGELBPD ++ // Default to TAGE-L BPD
3      new Config((site, here, up) => {
4        case TilesLocated(InSubsystem) => {
5          val prev = up(TilesLocated(InSubsystem), site)
6          val idOffset = overrideIdOffset.getOrElse(prev.size)
7          (0 until n).map { i =>
8            BoomTileAttachParams(
9              tileParams = BoomTileParams(
10               core = BoomCoreParams(
11                 fetchWidth = 8,
12                 decodeWidth = 3,
13                 numRobEntries = 96,
14                 issueParams = Seq(
15                   IssueParams(issueWidth=1, numEntries=16, iqType=
     IQT_MEM.litValue, dispatchWidth=3),
16                   IssueParams(issueWidth=3, numEntries=32, iqType=
     IQT_INT.litValue, dispatchWidth=3),
17                   IssueParams(issueWidth=1, numEntries=24, iqType=
     IQT_FP.litValue , dispatchWidth=3)),
18                 numIntPhysRegisters = 100,
19                 numFpPhysRegisters = 96,
20                 numLdqEntries = 24,
21                 numStqEntries = 24,
22                 maxBrCount = 16,
23                 numFetchBufferEntries = 24,
24                 ftq = FtqParameters(nEntries=32),
25                 fpu = Some(freechips.rocketchip.tile.FPUParams(
     sfmaLatency=4, dfmaLatency=4, divSqrt=true))
26             ),
```

```
27                dcache = Some(
28                  DCacheParams(rowBits = site(SystemBusKey).beatBits,
    nSets=64, nWays=8, nMSHRs=4, nTLBWays=16)
29                ),
30                icache = Some(
31                  ICacheParams(rowBits = site(SystemBusKey).beatBits,
    nSets=64, nWays=8, fetchBytes=4*4)
32                ),
33                hartId = i + idOffset
34              ),
35              crossingParams = RocketCrossingParams()
36            )
37          } ++ prev
38        }
39        case SystemBusKey => up(SystemBusKey, site).copy(beatBytes =
    16)
40        case XLen => 64
41      })
42    )
```

6. Arquitectura homogénea multicore con dos procesadores Boom.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
1     class DualBoomConfig extends Config(
2       new boom.common.WithNLargeBooms(2) ++
3       new chipyard.config.AbstractConfig)
```

   Observamos que la definición de la clase es análoga a la anterior con la excepción de que el parámetro de número de cores que le pasamos a la clase *WithNLargeBooms* tiene el valor 2. Por tanto, generaremos una arquitectura con dos cores Boom con las mismas características que el procesador de la arquitectura unicore con el fin de poder comparar los resultados de las ejecuciones de las mismas aplicaciones sobre ambas.

7. Arquitectura heterogénea unicore con un procesador Boom y el coprocesador vectorial Hwacha. La configuración de esta arquitectura viene dada por la siguiente clase.

```
1     class HwachaLargeBoomConfig extends Config(
2       new chipyard.config.WithHwachaTest ++
3       new hwacha.DefaultHwachaConfig ++
4       new boom.common.WithNLargeBooms(1) ++
5       new chipyard.config.AbstractConfig)
```

   Esta arquitectura contiene un procesador Boom *Large*, el módulo del coprocesador vectorial Hwacha con la misma configuración que se ha visto anteriormente para la arquitectura heterogénea con procesador Rocket y el módulo de tests de Hwacha.

8. Arquitectura heterogénea unicore con un procesador Boom y el acelerador de multiplicación de matrices Gemmini.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
class GemminiBoomConfig extends Config(
new gemmini.DefaultGemminiConfig ++
new boom.common.WithNLargeBooms(1) ++
new chipyard.config.AbstractConfig)
```

Esta arquitectura contiene un procesador Boom *Large* y el módulo del acelerador Gemmini con la misma configuración que se ha visto anteriormente para la arquitectura heterogénea con procesador Rocket y el acelerador Boom.

9. Arquitectura heterogénea dualcore con un procesador Rocket y un Boom y el coprocesador vectorial Hwacha.

   La configuración de esta arquitectura viene dada por la siguiente clase.

```
class HwachaLargeBoomAndHwachaRocketConfig extends Config(
  new chipyard.config.WithHwachaTest ++
  new hwacha.DefaultHwachaConfig ++
  new boom.common.WithNLargeBooms(1) ++
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
```

Notamos que la arquitectura está compuesta por un procesador Rocket *Big*, un procesador Boom *Large*, el coprocesador vectorial Hwacha y el módulo de los tests de este último. Esta arquitectura está generada adhoc para el escenario de IoT en el que trabajamos, ya que necesitamos un procesador potente como el Boom *Large* y el coprocesador Hwacha para realizar los cómputos sobre las imágenes, y en paralelo, necesitamos el procesador Rocket para realizar envíos de información a la nube, como las imágenes captadas y las líneas detectadas para realizar posteriores análisis sobre estos datos.

Así pues, hemos visto en detalle los diseños de las arquitecturas que necesitábamos y siguiendo los pasos de la sección 4.1 se ha generado para cada uno de ellos una AGFI que podremos volcar sobre una FPGA para simular todos los componentes de nuestro diseño fielmente. Pasemos entonces a ver la generación de los *workload* que utilizaremos para el testeo de los diseños y la obtención de resultados.

4.3. Generación de los *workload*

En esta sección vamos a ver el procedimiento a seguir para generar los ficheros necesarios para que se ejecuten las aplicaciones deseadas sobre los diseños generados anteriormente. Con el fin de comparar la ejecución de las aplicaciones en distintas arquitecturas y ver el rendimiento obtenido, se añadirán lecturas a los contadores hardware que nos ofrece RISC-V para recoger las siguientes métricas sobre cada ejecución: ciclos e instrucciones retiradas.

En primer lugar, vamos a ver el proceso que se sigue para generar un *workload* que ejecute una aplicación. Esta parte de compilación de una aplicación se va a realizar con la herramienta FireMarshal. Para ello, nos dirigimos al directorio `firesim/sw/firesim-software` donde encontramos el ejecutable FireMarshal, así como las carpetas auxiliares que necesitamos para ordenar los programas a ejecutar.

Una vez en este directorio, nos dirigimos a la carpeta `/tests` donde crearemos un fichero .json de configuración con el nombre de nuestra aplicación a ejecutar, por ejemplo, hello-world.json, así como una carpeta hello-world donde incluiremos el código fuente y otros ficheros necesarios. Ahora, completaremos el fichero .json con los valores de configuración necesarios para que FireMarshal pueda generar el ejecutable completo de nuestra aplicación. Veamos primero algunas de las opciones más relevantes que puede contener este fichero de configuración, que se pueden consultar en [31].

- Name: Especifica el nombre del *workload* que se utilizará para nombrar los ficheros derivados que se generen.
- Base: Especifica la configuración base sobre la que se montará la aplicación. Esta configuración permite elegir entre tres opciones: bare, para no utilizar sistema operativo; br-base, para usar la distribución buildroot de linux, más información en [10] [11]; y, fed-base para la distribución de fedora de Linux, más información en [12] [13].
- Host-init: Especifica un script que se ejecutará en el host cada vez que se genere el *workload*.
- Guest-init: Especifica un script que se ejecutará en el guest cuando se genere el ejecutable. Este script se ejecuta desde la ruta `/root/` y tiene permisos de administrador, por tanto, se podrían instalar las dependencias que necesite el programa a ejecutar. Es por esto que debe terminar con una llamada al comando poweroff para que apague la máquina y posteriormente se inicie, para que los cambios que se hayan podido producir queden bien instalados y referenciados.
- Overlay: Estructura de ficheros y carpetas que se copiará a la ruta `/root/` de la instancia que se inicie. Se utiliza principalmente para sobrescribir ficheros de configuración del sistema, como por ejemplo, `/etc/fstab`.
- Post-bin: Especifica un script que se ejecutará en el host después de generar los binarios pero antes de que se generen las imágenes de las aplicaciones.
- Outputs: Especifica una lista de rutas absolutas de ficheros que se copiarán a la instancia host al terminar de ejecutar el *workload*.
- Run: Especifica un script que se ejecuta automáticamente cada vez que se ejecute este *workload*. Si se desea que la ejecución del workload sea automática y la instancia y la simulación finalicen al terminar, se debe incluir una llamada al comando poweroff al final del script. En caso contrario, se deberá acceder vía ssh o mosh a la instancia y ejecutar el poweroff.
- Bin: Especifica una ruta absoluta al fichero binario que se utilizará para ejecutar el *workload*. Esta opción es especialmente útil para los programas que se ejecuten sobre *bare-metal* ya que la propia compilación en el host genera un binario con la configuración de arranque de la máquina.
- Workdir: Especifica el directorio a utilizar para los ficheros fuente del *workload*.
- Jobs: Especifica una lista de trabajos que se ejecutarán secuencialmente en la máquina destino. Esto permite ejecutar distintas aplicaciones utilizando el mismo workload. Además, dentro de la especificación de cada trabajo se pueden utilizar otras opciones para indicar distintas configuraciones para cada trabajo.

Vistas algunas de las opciones más relevantes que existen, pasamos a ver unos ejemlos que esclarezcan su uso. En primer lugar, supongamos que nuestra aplicación *hello-world* se va a ejecutar sobre un entorno sin sistema operativo. Entonces el fichero de configuración es bastante sencillo pues solo contendría las opciones *name* y *base* que son obligatorias y la opción *bin* para indicar el binario a ejecutar. A continuación, lo vemos.

```
{
  "name" : "hello-world",
  "base" : "bare",
  "bin" : "hello-world.riscv"
}
```

En cuanto al directorio `/test/hello-world`, este debería contener el ejecutable *hello-world.riscv* en la raíz del directorio y además, puede contener subdirectorios con el código fuente necesario para generar dicho ejecutable.

En segundo lugar, supongamos que tenemos un *workload* que utilizará la distribución de linux buildroot y que debe realizar algunas instalaciones en la máquina de destino antes de poder ejecutarse. Entonces necesitaremos un fichero de configuración *ejemplo.json* que contenga además de las opciones estrictamente necesarias, la opción *guest-init* para indicar el script a ejecutar en el guest y la opción *run* para indicar el script que se ejecutará como aplicación del *workload*. A continuación, veamos como quedaría el fichero de configuración.

```
{
  "name" : "ejemplo",
  "base" : "br-base.json",
  "guest-init": "/scripts/guest-init.sh",
  "run" : "/scripts/run.sh"
}
```

En esta ocasión, el directorio */test/ejemplo* debería contener un subdirectorio *scripts* con los scripts que hemos indicado y, opcionalmente, contendrá el código fuente del ejemplo, el Makefile para la compilación y el ejecutable que será llamado desde el script *run.sh*.

Ahora, una vez escrita la configuración en el fichero .json[1] pasamos a generar e instalar el *workload*. Para ello, primero invocamos el ejecutable de FireMarshal con la opción **build** y el nombre del fichero .json que contiene la configuración del *workload*, con lo que se generan los ficheros rootfs, *[workload-name].img*, y los binarios de arranque, *[workload-name]-bin*, en el directorio `/images`.

Finalmente, faltaría llevar dichos ficheros a los directorios de FireSim y crear un fichero de configuración .json para FireSim. Para ello, invocamos nuevamente el ejecutable FireMarshal con la opción *install* y el nombre del *workload*, de forma que se generan los ficheros .json de configuración donde si indican los binarios a utilizar en la instancia que se inicia y los *workload* que se ejecutan en cada nodo. A continuación, presentamos el fichero de configuración de un ejemplo sencillo que ejecuta el mismo *workload* en todos los nodos.

[1] En versiones nuevas del repositorio también se acepta el formato .yaml para los ficheros de configuración.

```
{
  "benchmark_name"            : "linux-uniform",
  "common_bootbinary"         : "br-base-bin",
  "common_rootfs"             : "br-base.img",
  "common_outputs"            : ["/etc/os-release"],
  "common_simulation_outputs" : ["uartlog", "memory_stats*.csv"]
}
```

A pesar de que este fichero es editable y existen más opciones además de las vistas en el ejemplo, no se detalla su funcionamiento en la documentación, ya que se recomienda encarecidamente la utilización de la opción *install* de FireMarshal para generar automáticamente estos ficheros en las rutas correspondientes.

Llegados a este punto, estamos en disposición de escribir los ficheros de configuración necesarios para conseguir *workloads* que ejecuten las aplicaciones deseadas, así como, instalarlos para obtener la configuración necesaria para ejecutarlos en FireSim. Pasemos entonces a ver los *workload* que hemos generado.

4.4. *Workloads* generados

1. *Workload* 1: Aplicación multihilo sobre la distribución buildroot de linux.

   En este *workload* se ejecutará una aplicación multihilo creada adhoc para explotar la paralelización de los hilos. En particular, hemos generado una aplicación cuyos hilos ejecutan un número fijo N de sumas con valores independientes generados aleatoriamente previamente con el código siguiente. Además, el bucle de sumas se repite N_times veces para que la aplicación consuma más tiempo al ejecutarse y se puedan ver mejores resultados.

```
    void *myThreadFun(void *vargp)
    {
      int a,b,c;
      int * nThread = (int *) vargp;
      for(int j=0; j < N_times; j++){
        for (int i=0; i < N; i++){
          a = rand_A_vals[i + *nThread * N];
          b = rand_B_vals[i + *nThread * N];
          c = a + b;
          if (i%vivo == 0) {
            printf("En proceso...%d\n", i);
          }
        }
      }
    }
```

   Ahora que hemos presentado el código de la aplicación nos queda la compilación y la generación del fichero .json de configuración para poder generarlo e instalarlo con FireMarshal. Dado que se va a ejecutar sobre la distribución buildroot de linux, compilaremos el ejemplo el siguiente fichero *Makefile*.

```
CC=riscv64-unknown-linux-gnu-gcc
CFLAGS=-O2 -static -march=rv64gc
LDFLAGS=-static -lgcc -lm -lpthread -lgcc -march=rv64gc
OBJS = main.c \

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@

all: hello-thread
hello-thread: $(OBJS)
        $(CC) -o hello-thread $(OBJS) $(LDFLAGS)

clean:
        rm -f *.o hello-thread
```

Observamos que utilizamos el compilador de gcc especial para el repertorio RISC-V de 64 bits y para linux, cuyo ejecutable se llama *riscv64-unknown-linux-gnu-gcc*. Y en la opción *-march* le indicamos el repertorio de instrucciones que utilizamos.

Ahora, nos encargamos del entorno de FireMarshal. En primer lugar, en el directorio `/firesim/sw/firesim-software/tests` creamos un directorio llamado *hello-thread*. A continuación, dentro de ese directorio, creamos dos scripts: *run.sh* y *build.sh*, y un directorio */overlay* que contendrá el código fuente de la aplicación y el Makefile. El script *build.sh* se limita a ejecutar el comando *make* en el directiorio `/overlay`, y el script *run.sh* llama al binario *hello-thread* que existirá en el directorio `/overlay` y ejecuta el comando *poweroff*.

En cuanto al fichero de configuración de FireMarshal, a continuación presentamos su contenido.

```
{
  "name" : "hello-thread",
  "base" : "br-base.json",
  "host-init" : "build.sh",
  "overlay" : "overlay",
  "run" : "run.sh"
}
```

2. *Workload* 2: Algoritmo de detección de líneas sobre la distribución buildroot de linux.

En este *workload*, queremos ejecutar el código de detección de imágenes con los tipos de datos modificados que presentábamos en el Capítulo 3. Para poder ejecutarlo usando el repertorio de instrucciones RISC-V y ejecutarlo sobre las arquitecturas unicore con procesador Rocket o Boom necesitamos adaptarlo ligeramente.

En particular, debemos eliminar las librerías de tiempos y la función para leer el tiempo desde la librería, ya que en RISC-V contamos con dos registros hardware que nos proporcionan los ciclos y las instrucciones retiradas. Y en el caso de ejecutar sobre un sistema operativo, como es este caso, disponemos también de un registro que nos permite leer el tiempo. Así pues, usaremos las siguientes funciones para leer las tres métricas.

```
unsigned long read_cycles(void)
{
  unsigned long cycles;
  asm volatile ("rdcycle %0" : "=r" (cycles));
  return cycles;
}
unsigned long read_time(void)
{
  unsigned long time;
  asm volatile ("rdtime %0" : "=r" (time));
  return time;
}
unsigned long read_instret(void)
{
  unsigned long instret;
  asm volatile ("rdinstret %0" : "=r" (instret));
  return instret;
}
```

Además, no solo realizaremos las lecturas de las métricas al inicio y al final de toda la ejecución de los algoritmos de tratamiento de imágenes, sino que tomaremos las métricas para cada uno de los algoritmos que toman parte en el tratamiento de la imagen: el algoritmo de Canny, transformada de Hough y la obtención de líneas, pues hemos visto en el *profiling* del código que estas partes son las más críticas en tiempo.

Ahora que hemos presentado el código de la aplicación nos queda la compilación y la generación del fichero .json de configuración para poder generarlo e instalarlo con FireMarshal. Dado que se va a ejecutar sobre la distribución buildroot de linux, compilaremos el ejemplo con un fichero *Makefile* análogo al del anterior *workload*. Utilizando el compilador de gcc especial para el repertorio RISC-V de 64 bits y para linux e indicando en la opción *-march* el mismo repertorio de instrucciones.

Ahora, nos encargamos del entorno de FireMarshal. En primer lugar, en el directorio `/firesim/sw/firesim-software/tests` creamos un directorio llamado *line-detection*. A continuación, dentro de ese directorio creamos un script *run.sh* y el directorio */overlay/src* que contendrá el código fuente de la aplicación y el Makefile. El script *run.sh* llama al binario *line-detection* que existirá en el directorio `/overlay/src` y ejecuta el comando *poweroff*.

En cuanto al fichero de configuración de FireMarshal, a continuación presentamos su contenido.

```
{
  "name" : "line-detection",
  "base" : "br-base.json",
  "overlay" : "overlay",
  "run" : "run.sh"
}
```

3. *Workload* 3: Algoritmo de detección de líneas utilizando Gemmini sobre *bare-metal*.

   En este *workload*, queremos ejecutar el código de detección de imágenes que hemos adaptado en el *workload* 2 sobre una arquitectura con el acelerador de multiplicación de matrices Gemmini. Para ello, debemos sustituir las multiplicaciones de matrices que se realizan elemento a elemento por una llamada a una multiplicación en Gemmini.

Para ello, utilizaremos el hecho de que Gemmini cuenta con macros en C para realizar las operaciones necesarias para multiplicar dos matrices, como son mover datos entre la memoria principal del procesador y la memoria scratchpad de Gemmini, definir el procesamiento por bloques de la matriz, etc. En nuestro caso, utilizaremos la siguiente macro:

```
1    static void tiled_matmul_auto(size_t dim_I, size_t dim_J, size_t
     dim_K,
2        const elem_t* A, const elem_t* B,
3        const void * D, void * C,
4        size_t stride_A, size_t stride_B, size_t stride_D, size_t
     stride_C,
5        scale_t A_scale_factor, scale_t B_scale_factor, scale_acc_t
     D_scale_factor,
6        int act, acc_scale_t scale, size_t relu6_shift, bool
     repeating_bias,
7        bool transpose_A, bool transpose_B,
8        bool full_C, bool low_D,
9        uint8_t weightA,
10       enum tiled_matmul_type_t tiled_matmul_type)
```

A esta función se le pasan los siguientes parámetros: las dimensiones de las matrices de entrada y de salida; las propias matrices de entrada; los factores de escalado que en nuestro caso tendrán el valor 1, pues no queremos escalar las matrices; los factores de stride, que en nuestro caso tendrán el valor 0 pues nuestras matrices son de tamaño 5x5 y Gemmini tiene un array sistólico de 16x16; los booleanos para indicar si se trasponen, con el valor False en nuestro caso; y otros parámetros de configuración a los que les daremos valores por defecto definidos en ficheros propios de Gemmini, como *gemmini_params.h* o *gemmini_testutils.h*.

A continuación, creamos el directorio `line-detection-gemmini` en los tests de FireMarshal y lo configuramos análogo al *workload* 2. También el fichero de configuración .json es análogo al de este *workload* cambiando el nombre del ejemplo. Con esto, tenemos el *workload* preparado para compilarse e instalarse usando FireMarshal.

4.5. Ejecución de una simulación

En esta sección, vamos a utilizar las imágenes de las arquitecturas que hemos diseñado y generado en las secciones 4.1 y 4.2, así como los *workload* que hemos definido y generado en las secciones 4.3 y 4.4, para explicar el proceso a seguir en una ejecución de una simulación y obtener sus resultados.

En primer lugar, nos dirigimos al directorio `/firesim/deploy` donde encontramos el ejecutable de FireSim, así como los directorios `/results-build` y `/results-workload`. En este último directorio se almacenarán los resultados de las simulaciones, mientras que en el primero ya tenemos almacenadas las imágenes y los ficheros necesarios que se han creado para nuestros *workload* cuando hemos usado el comando *install* de FireMarshal.

Llegados a este punto, el único fichero de configuración que nos resta modificar es *config_runtime.ini*. En este fichero, se encuentra el nombre de la granja que se inicia, la configuración relacionada con el número y tipo de instancias que se iniciarán en la simulación, la configuración de la topología de red entre los nodos si la hay, el diseño de

la arquitectura que se va a utilizar, el *workload* que se va a a ejecutar y algunas opciones adicionales propias de AWS. A continuación, vemos el fichero que se presenta por defecto en el repositorio.

```
# RUNTIME configuration for the FireSim Simulation Manager
# See docs/Advanced-Usage/Manager/Manager-Configuration-Files.rst for
# documentation of all of these params.

[runfarm]
runfarmtag=mainrunfarm

f1_16xlarges=1
m4_16xlarges=0
f1_4xlarges=0
f1_2xlarges=0

runinstancemarket=ondemand
spotinterruptionbehavior=terminate
spotmaxprice=ondemand

[targetconfig]
topology=example_8config
no_net_num_nodes=2
linklatency=6405
switchinglatency=10
netbandwidth=200
profileinterval=-1

# This references a section from config_hwconfigs.ini
# In homogeneous configurations, use this to set the hardware config
    deployed
# for all simulators
defaulthwconfig=firesim-rocket-quadcore-nic-l2-llc4mb-ddr3

[tracing]
enable=no

# Trace output formats. Only enabled if "enable" is set to "yes" above
# 0 = human readable; 1 = binary (compressed raw data); 2 = flamegraph (
    stack
# unwinding -> Flame Graph)
output_format=0

# Trigger selector.
# 0 = no trigger; 1 = cycle count trigger; 2 = program counter trigger; 3 =
# instruction trigger
selector=1
start=0
end=-1

[autocounter]
readrate=0

[workload]
workloadname=linux-uniform.json
terminateoncompletion=no
suffixtag=

[hostdebug]
# When enabled (=yes), Zeros-out FPGA-attached DRAM before simulations
```

```
# begin (takes 2-5 minutes).
# In general, this is not required to produce deterministic simulations on
# target machines running linux. Enable if you observe simulation non-determinism.
zerooutdram=no
# If disable_synth_asserts=no, simulation will print assertion message and
# terminate simulation if synthesized assertion fires.
# If disable_synth_asserts=yes, simulation ignores assertion firing and
# continues simulation.
disable_synth_asserts=no

[synthprint]
# Start and end cycles for outputting synthesized prints.
# They are given in terms of the base clock and will be converted
# for each clock domain.
start=0
end=-1
# When enabled (=yes), prefix print output with the target cycle at which the print was triggered
cycleprefix=yes
```

En cada ejecución, tendremos que modificar algunos de los parámetros que vienen en el fichero. Sin embargo, como las arquitecturas que hemos generado comparten características básicas, algunos de ellos solo los modificaremos en la primera ejecución. Veamos en detalle qué parámetros modificamos y en qué ejecuciones.

- Runfarm: En este apartado viene por defecto una instancia del tipo *f1.16xlarge*, sin embargo, para nuestras arquitecturas será suficiente con una instancia del tipo *f1.2xlarge*. Por tanto, debemos indicar 1 en la instancia de tipo *f1_ 2xlarges* y 0 en las instancias restantes.

- Targetconfig: En este apartado se configura el número de nodos y la topología de red entre ellos. En nuestro caso, solo necesitamos un nodo en cada simulación y ninguna red. Por tanto, debemos modificar *no_ net_ num_ nodes* y darle valor 1, y en el campo *topology* asignar la opción *no_ net_ config*. Además, se elige el diseño de hardware que se va a utilizar mediante la opción *defaulthwconfig*. Esta opción la modificaremos en cada simulación, eligiendo el diseño adecuado.

- Workload: En este apartado se elige el *workload* que se va a ejecutar sobre la arquitectura mediante la opción *workloadname*. Esta opción también se modificará en cada simulación para indicar el *workload* a ejecutar.

  Además, existe la opción *terminateoncompletion* que indica si la instancia se parará y eliminará una vez se termine de ejecutar el *workload*. Si queremos tener las simulaciones completamente automatizadas y que no requieran de ninguna acción manual para terminar, debemos indicar que sí. Sin embargo, si queremos realizar pruebas dónde nos conectemos a la instancia al acabar la simulación para observar algún elemento de la salida, podemos indicar que no. En este segundo caso, deberemos apagar la instancia manualmente conectándonos y ejecutando el comando *poweroff*.

Ahora, para ejecutar una simulación completa, se debe invocar al ejecutable *firesim* con unas opciones específicas en un orden específico. Veamos los comandos a ejecutar a continuación:

1. Launchrunfarm: Esta opción iniciará tantas instancias de cada tipo como le hayamos indicado previamente en el fichero de configuración .ini. La salida resultante de la ejecución del comando indicará el número de instancias que se han iniciado de cada tipo así como su identificador. Veamos a continuación, un ejemplo de la salida esperada.

```
FireSim Manager. Docs: http://docs.fires.im
Running: launchrunfarm

Waiting for instance boots: 0 f1.16xlarges
Waiting for instance boots: 0 f1.4xlarges
Waiting for instance boots: 0 m4.16xlarges
Waiting for instance boots: 1 f1.2xlarges
i-0d6c29ac507139163 booted!
The full log of this run is:
/home/centos/firesim/deploy/logs/22022-01-13--11-01-37-launchrunfarm-456INT48SVXBEA2F.log
```

2. Infrasetup: Esta opción se encarga de volcar sobre la FPGA la AGFI de la arquitectura y de instalar el resto de componentes software que se necesiten para ejecutar el *workload*. Este proceso tarda unos minutos y la salida que se produce debería ser similar a la siguiente.

```
FireSim Manager. Docs: http://docs.fires.im
Running: infrasetup

Building FPGA software driver for FireSim-DDR3FRFCFSLLC4MB_WithDefaultFireSimBridges_WithFireSimTestChip
ConfigTweaks_chipyard.DualBoomConfig-MCRams_F80MHz_BaseF1Config
[192.168.4.26] Executing task 'instance_liveness'
[192.168.4.26] Checking if host instance is up...
[192.168.4.26] Executing task 'infrasetup_node_wrapper'
[192.168.4.26] Copying FPGA simulation infrastructure for slot: 0.
[192.168.4.26] Installing AWS FPGA SDK on remote nodes. Upstream hash: 6c707ab4a26c2766b916dad9d40727266fa0e4ef
[192.168.4.26] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[192.168.4.26] Copying AWS FPGA XDMA driver to remote node.
[192.168.4.26] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[192.168.4.26] Loading XDMA Driver Kernel Module.
[192.168.4.26] Setting up remote node for qcow2 disk images.
[192.168.4.26] Unloading NBD Kernel Module.
[192.168.4.26] Disconnecting all NBDs.
[192.168.4.26] Loading NBD Kernel Module.
[192.168.4.26] Clearing FPGA Slot 0.
[192.168.4.26] Checking for Cleared FPGA Slot 0.
[192.168.4.26] Flashing FPGA Slot: 0 with agfi: agfi-0707524a52f7f68e4.
[192.168.4.26] Checking for Flashed FPGA Slot: 0 with agfi: agfi-0707524a52f7f68e4.
[192.168.4.26] Unloading XDMA/EDMA/XOCL Driver Kernel Module.
[192.168.4.26] Loading XDMA Driver Kernel Module.
[192.168.4.26] Starting Vivado hw_server.
[192.168.4.26] Starting Vivado virtual JTAG.
The full log of this run is:
/home/centos/firesim/deploy/logs/2022-01-13--11-03-59-infrasetup-4UVLJAR5GF8BTM2C.log
```

3. Runworkload: Esta opción empieza la ejecución del *workload* correspondiente. En primer lugar, la salida muestra las inicializaciones que se realizan, como vemos a continuación:

```
FireSim Manager. Docs: http://docs.fires.im
Running: runworkload

Creating the directory: /home/centos/firesim-new/deploy/results-workload/2018-05-19--00-38-52-linux-uniform/
[172.30.2.174] Executing task 'instance_liveness'
[172.30.2.174] Checking if host instance is up...
[172.30.2.174] Executing task 'boot_simulation_wrapper'
[172.30.2.174] Starting FPGA simulation for slot: 0.
[172.30.2.174] Executing task 'monitor_jobs_wrapper'
```

A continuación, se reemplaza el contenido de la consola por actualización periódicas que muestran el progreso de la simulación, como podemos ver a continuación:

```
This workload's output is located in:
/home/centos/firesim/deploy/results-workload/2022-01-13--11-08-41-hello-thread/
This run's log is located in:
/home/centos/firesim/deploy/logs/2022-01-13--11-08-41-runworkload-IB2OR7S7BNQKWFGK.log
This status will update every 10s.
------------------------------------------------
Instances
------------------------------------------------
Instance IP:   192.168.4.26 | Terminated: False
------------------------------------------------
Simulated Switches
------------------------------------------------
------------------------------------------------
Simulated Nodes/Jobs
------------------------------------------------
Instance IP:   192.168.4.26 | Job: hello-thread0 | Sim running: True
------------------------------------------------
Summary
------------------------------------------------
1/1 instances are still running.
1/1 simulations are still running.
------------------------------------------------
```

Una vez hemos llegado a este punto y la instancia está en funcionamiento, si queremos, podemos conectarnos a ella y ver la evolución real de la ejecución del *workload*. Para ello, basta con conectar a la IP local que se indica, en este caso 192.168.4.26, vía ssh desde otra consola del administrador principal y ejecutar el comando "screen -r fsim0" para que se conecte a la salida que está generando la ejecución del *workload*. Sin embargo, si el *workload* está preparado para capturar métricas como ciclos, tiempos o instrucciones retiradas, esta conexión puede generar no-determinismo en las lecturas, ya que supone una carga extra para el procesador.

4. Terminaterunfarm: Esta opción se encarga de comprobar las instancias activas que hay asociadas a la granja que se ha iniciado anteriormente. En caso de que quede

alguna activa por un mal funcionamiento de la simulación o por elección en la configuración, permite parar y eliminar la instancia desde la propia consola. Sin embargo, no asegura completamente que se pueda eliminar la instancia desde la propia línea de comandos de FireSim y se recomienda comprobarlo en el panel de AWS. La salida a esperar en este caso, sería la siguiente:

```
FireSim Manager. Docs: http://docs.fires.im
Running: terminaterunfarm

IMPORTANT!: This will terminate the following instances:
f1.16xlarges
[]
f1.4xlarges
[]
m4.16xlarges
[]
f1.2xlarges
['i-01a2b92ef6870b592']
Type yes, then press enter, to continue. Otherwise, the operation will be cancelled.
yes
Instances terminated. Please confirm in your AWS Management Console.
The full log of this run is:
/home/centos/firesim/deploy/logs/2022-01-13--11-09-51-terminaterunfarm-E87G9IW9CYWK5IPA.log
```

Llegados a este punto, tenemos las arquitecturas necesarias y toda la información necesaria para realizar experimentos en los que simularemos la ejecución de un *workload* específico sobre varias arquitecturas con el fin de extraer resultados.

4.6. Experimentos

En esta sección, vamos a simular la ejecución de los *workload* generados anteriormente sobre distintas arquitecturas también generadas previamente, para obtener resultados de las métricas de ciclos, tiempo e instrucciones retiradas, con el fin de elegir la arquitectura idónea para cada caso de uso.

4.6.1. Experimento 1: Ejecución de una aplicación multihilo sobre arquitecturas unicore y dualcore con el procesador Rocket y Boom

En este experimento, vamos a explorar la eficacia de las aplicaciones paralelizables. Para ello, utilizamos el *workload* 1 que ejecuta una aplicación adhoc con tantos hilos independientes como cores tenga la arquitectura, en este caso, dos. Una vez elegido el *workload* que se va a ejecutar, veamos las arquitecturas sobre las que se va a ejecutar.

En primer lugar, tenemos dos arquitecturas sencillas unicores con el procesador Rocket y Boom respectivamente, y con las opciones de memoria, mejoras de memoria y frecuencia de reloj a 80MHz vistas anteriormente. En segundo lugar, tenemos dos arquitecturas de dos cores cada una con los procesadores Rocket y Boom respectivamente, y las mismas

	Ciclos	Tiempo(us)	Instr. retiradas
Rocket singlecore	2013568309	1258481	971588270
Boom singlecore	917857436	573660	965394569
Rocket dualcore	997358541	643723	480749599
Boom dualcore	453319676	299234	471682168
Speedup Boom vs Rocket	x2,19377022	x2,193775058	x1,00641572
Speedup Rocket dual vs single	x2,018901154	x1,955003938	x2,020986127
Speedup Boom dual vs single	x2,024746519	x1,917094983	x2,046705673

Tabla 4.1: Resultados de tiempo, ciclos e instrucciones retiradas en el experimento multihilo para $N\_times = 1$.

	Ciclos	Tiempo(us)	Instr. retiradas
Rocket singlecore	7979910236	4987444	3872031120
Boom singlecore	3591056463	2288162	3861080877
Rocket dualcore	3999152447	2515621	1931257314
Boom dualcore	1809721983	1154538	1913436432
Speedup Boom vs Rocket	x2,222162285	x2,179672593	x1,002836056
Speedup Rocket dual vs single	x1,995400361	x1,982589587	x2,004927615
Speedup Boom dual vs single	x1,984313887	x1,981885395	x2,017877789

Tabla 4.2: Resultados de tiempo, ciclos e instrucciones retiradas en el experimento multihilo para $N\_times = 4$.

configuraciones adicionales que las arquitecturas unicore para conseguir una comparación de tiempos justa.

Ahora, en la Tabla 4.1 vemos los resultados de tiempo, ciclos e instrucciones retiradas que se han obtenido para cada una de las arquitecturas en una primera ejecución de la aplicación con la variable $N\_times = 1$, que indica que el bucle de sumas se ejecuta tan solo una vez. Podemos observar como el *speedup* de Boom respecto Rocket es notable, ya que el procesador Boom ejecuta instrucciones fuera de orden y, por tanto, es más rápido ejecutando el mismo número de instrucciones. En cuanto al *speedup* de las arquitecturas dualcore respecto a la unicore obtenemos un valor de x2 en ciclos como esperábamos ya que la aplicación utiliza 2 hilos y cada uno de ellos es completamente independiente. Sin embargo, en el tiempo el *speedup* no llega a ese valor y se queda en x1.95 o menos, esto se debe a que el tiempo total que consume la aplicación es muy pequeño, estando por debajo de 1.5 segundos, por tanto, cualquier variación en el entorno puede influir produciendo un incremento de tiempo.

A continuación, para obtener un valor de *speedup* en la métrica de tiempo más ajustado a 2, vamos a ejecutar la aplicación para distintos valores de N_times. En particular, para $N\_times = 4$ y $N\_times = 8$. En las Tablas 4.2 y 4.3 podemos comprobar como el *speedup* de la métrica del tiempo se acerca al valor x2 cuanto mayor es el número de ejecuciones, obteniendo x1.99 con $N\_times = 8$.

De esta forma, queda comprobado que las aplicaciones multicore se ejecutan correctamente sobre estas arquitecturas dualcore y se consigue el *speedup* esperado. Por tanto, en escenarios en los que ejecutemos distintas aplicaciones independientes o aplicaciones altamente paralelizables nos será de gran utilidad generar arquitecturas multicore sobre

	Ciclos	Tiempo(us)	Instr. retiradas
Rocket singlecore	15940000765	9962500	7739246099
Boom singlecore	7316660515	4572912	7721902943
Rocket dualcore	7990401234	5005957	3867108941
Boom dualcore	3662766796	2297044	3859136000
Speedup Boom vs Rocket	x2,178589636	x2,178589923	x1,002245969
Speedup Rocket dual vs single	x1,99489366	x1,99012896	x2,001300252
Speedup Boom dual vs single	x1,997577493	x1,990781195	x2,000940869

Tabla 4.3: Resultados de tiempo, ciclos e instrucciones retiradas en el experimento multihilo para $N\_times = 8$.

las que ejecutarlo.

4.6.2. Experimento 2: Ejecución de la aplicación de detección de líneas sobre arquitecturas unicore con el procesador Rocket o Boom

En este experimento, simulamos la ejecución del *workload* 2 sobre las arquitecturas unicore homogéneas con Rocket y Boom a una frecuencia de 80MHz para obtener unos valores de ciclos, tiempo e instrucciones retiradas que utilizaremos como referencia para elegir el procesador más adecuado.

En la Tabla 4.4 vemos los resultados de tiempo, ciclos e instrucciones retiradas que se han obtenido para cada una de las arquitecturas. Notamos que la transformada de Hough es una parte poco relevante en cuanto a rendimiento, ya que ni siquiera el procesador Boom es capaz de mejorar sus métricas, esto quiere decir, que las operaciones que ejecuta dependen de las anteriores y no se mejora el tiempo con la posibilidad de ejecutar instrucciones fuera de orden. También, observamos que el algoritmo para obtener las líneas obtiene un *speedup* de x2 en el procesador Boom respecto al Rocket, sin embargo, también es poco relevante respecto al total del programa, ya que el tiempo y los ciclos que consume son pocos.

Finalmente, observamos que el algoritmo de Canny es la parte del programa más pesada en cuanto a tiempo y ciclos y conseguimos un *speedup* de x2 con el procesador Boom respecto al Rocket. Con lo que podemos deducir que es interesante utilizar el procesador Boom para la ejecución de este programa, ya que hay numerosas instrucciones que se pueden ejecutar fuera de orden y, por tanto, mejorar el rendimiento global de la ejecución.

4.6.3. Experimento 3: Ejecución de la aplicación de detección de líneas sobre arquitecturas heterogéneas con el procesador Rocket o Boom y el acelerador de multiplicación de matrices Gemmini

En este experimento, simulamos la ejecución del *workload* 3 sobre las arquitecturas unicore heterogéneas con Rocket o Boom y el acelerador de multiplicación matricial Gemmini a una frecuencia de 50MHz para obtener unos valores de ciclos, tiempo e instrucciones retiradas que utilizaremos como referencia para elegir el procesador más adecuado.

A continuación, en la Tabla 4.5 vemos los resultados de ciclos y tiempo obtenidos para esta simulación. [2] Estos resultados nos resultan un poco adversos en un primer vistazo, ya

[2]Notamos que no hemos medido las instrucciones retiradas porque este experimento se realizó sobre

		Ciclos	Tiempo(us)	Instr. retiradas
Rocket singlecore	Canny	2225197221	1390748	906346743
	Hough	331831187	207395	93537773
	GetLines	6502011	4065	3473559
BOOM singlecore	Canny	1080525975	675328	904879044
	Hough	316580283	197862	93534740
	GetLines	3232192	2021	3473993
Speedup Boom vs Rocket	Canny	x2,059364858	x2,059366708	x1,001621984
	Hough	x1,048173891	x1,048180045	x1,000032426
	GetLines	x2,011641326	x2,011380505	x0,9998750717

Tabla 4.4: Resultados de tiempo, ciclos e instrucciones retiradas en la simulación del algoritmo de detección de líneas sobre Linux.

		Canny	Hough	GetLines
Rocket singlecore	Ciclos	1.467.628,00	8.462,00	2.172,00
	Tiempo(us)	4.924.731.576,00	27.110.946,00	6.969.059,00
Boom singlecore	Ciclos	1.062.790,00	4.503,00	1.169,00
	Tiempo(us)	3.400.962.430,00	14.433.273,00	3.747.415,00
Rocket + Gemmini	Ciclos	606.748,00	87.848,00	2.091,00
	Tiempo(us)	1.941.728.496,00	281.215.543,00	6.768.720,00
Boom + Gemmini	Ciclos	347.759,00	95.141,00	1.080,00
	Tiempo(us)	1.112.954.401,00	304.543.560,00	3.517.131,00
Speedup Boom vs Rocket	Ciclos	x1,74	x0,92	x1,93
	Tiempo(us)	x1,74	x0,92	x1,92
Speedup Rocket + Gemmini vs Rocket single	Ciclos	x2,41	x0,096	x1,03
	Tiempo(us)	x2,53	x0,096	x1,02
Speedup Boom + Gemmini vs Boom single	Ciclos	x3,06	x0.05	x1,08
	Tiempo(us)	x3,06	x0,05	x1,07

Tabla 4.5: Resultados de tiempo y ciclos en la simulación del algoritmo de detección de lineas adaptado para Gemmini sobre Linux.

que las ejecuciones que utilizan el acelerador Gemmini no reflejan una notable disminución del tiempo y los ciclos, sino un *speedup* entre el x2 y x3. Esto es porque las matrices que estamos multiplicando en nuestra aplicación son de un tamaño más pequeño que el array sistólico de Gemmini, por tanto, en cada multiplicación no se aprovecha todo su potencial de cálculo, solo alrededor del 33 %. Esto no es sencillo de resolver, ya que la técnica de multiplicación por bloques de matrices nos puede aportar un ligero aumento del rendimiento, pero pequeño, ya que continuaríamos teniendo gran parte de elementos a 0 que no se utilizan en el cálculo final.

una versión del repositorio de FireSim antigua.

Capítulo 5

Conclusiones y Trabajo Futuro

En este capítulo presentaremos las conclusiones y las posibles líneas de trabajo futuro

5.1. Conclusiones

Las conclusiones extraídas de la elaboración de este trabajo, de más generales a más particulares, son las siguientes. En primer lugar, la arquitectura RISC-V, con su ecosistema hw/sw, aparece como una solución interesante, especialmente en el campo de IoT, gracias a que es *opensource* y a su modularidad y versatilidad. Permite desarrollar sistemas a medida para los distintos escenarios.

En segundo lugar, el entorno que hemos utilizado es potente y permite experimentar con distintas configuraciones, pero es muy complejo y está muy mal documentado. Una contribución importante de este trabajo es la documentación de los pasos necesarios para instalar y configurar el entorno, así como para generar distintos diseños y *workloads*.

Ya desde un punto de vista más específico, hemos comparado los resultados de ejecutar distintos algoritmos en procesadores con diferentes características, como son el Rocket y el Boom. Hemos podido comprobar que para algunos algoritmos no se nota mucha diferencia en tiempo de ejecución, mientras que para otros sí hay una diferencia significativa. Por eso, para los algoritmos de detección de líneas hemos optado por usar los procesadores más potentes.

También hemos ejecutado códigos en procesadores multicore, comprobando que la ejecución estaba repartiendo las tareas correctamente entre los núcleos y que la aceleración obtenida respecto de la ejecución en un único núcleo era la esperada.

También hemos generado diseños con aceleradores de propósito específico, como Gemmini y coprocesadores vectoriales como Hwacha y hemos conseguido ejecutar códigos correctamente en ambos.

Finalmente, por lo que se refiere a la ejecución de los algoritmos de detección de líneas, se han adaptado los algoritmos originales para intentar conseguir un menor tiempo de ejecución sin pérdida de precisión. También se han propuesto modificaciones para adaptarlos a la ejecución en los aceleradores.

La ejecución de los algoritmos de detección de líneas en sistemas multicore ha funcionado correctamente. Hemos comprobado que el procesador Boom alcanza tiempos de ejecución menores.

Sin embargo, la ejecución de los mismos sobre un sistema con un acelerador Gemmini no ha alcanzado los resultados de aceleración esperados. El conocimiento que podemos extraer del experimento 4.6.3 es que el acelerador Gemmini no se puede utilizar en multiplicaciones de matrices de dimensiones pequeñas, ya que se desaprovecha gran parte de su potencial y no se obtiene un incremento del rendimiento en la ejecución del programa principal. Sin embargo, para matrices de gran tamaño que podemos encontrar en entornos IoT donde se trabaja con redes neuronales, este acelerador se ajustaría a los requisitos del programa y se conseguiría un mayor rendimiento, como se puede ver en [16].

Finalmente, la ejecución sobre el coprocesador Hwacha, en las configuraciones que hemos probado, no finaliza correctamente debido a que el sistema no tiene suficiente memoria.

5.2. Trabajo futuro

Como ya se ha mencionado, en este trabajo hemos desarrollado una metodología y documentación que permitirán obtener nuevos diseños de manera sencilla. Esto abre nuevas vías a plantear otros problemas que se desee resolver, o a buscar soluciones mejores para los problemas planteados aquí.

Especialmente, queremos buscar otras aplicaciones, que puedan aprovechar el potencial del acelerador Gemmini y experimentar con otros diseños del coprocesador Hwacha, para intentar conseguir ejecutar en él la aplicación de detección de líneas.

Finalmente, nos gustaría complementar la aplicación de detección de líneas con algunas otras, de forma que se consiguiera un escenario de conducción autónoma más completo.

Chapter 6

Introduction

In this chapter we will make a brief introduction to the context in which this work is developed and the motivations that lead us to carry it out.

In the technological era in which we live, we strive every day to make all the usual tasks as automatic as possible in order to gain free time to spend it with the family or doing other activities. This is why the Internet of Things (IoT) arises, as we need new technologies to design these systems, usually embedded, that need a balance between low power consumption and high performance.

Some areas have already been extensively studied, such as the automation of production lines or home automation, as it is already common to find fully domotized houses that make our daily lives easier. However, in the land transport sector we still do not have these new developments.

In this area, the future seems to indicate that we will also have an intelligent, connected and autonomous transport model that makes use of artificial intelligence. If we dive into the technology that will be used, we realize that it brings together several areas. On the one hand, a great deal of computing power will be needed to execute many simultaneous operations that overlap in driving, for example detecting road lines to determine the trajectory and detecting obstacles or other vehicles, as well as recognizing traffic signs.

These actions are critical for driving safety, therefore, in addition to requiring high computational capacity, they need to be executed on-board, as even with a good network connection, the latency would be too large. However, other operations such as GPS tracking and the extraction of traffic data from it, can be sent and processed in the cloud, as they are not critical or do not require immediacy.

Given the number of critical and simultaneous operations that need to be executed for a good autonomous driving system, we need to design efficient and secure embedded SoCs with different processors and character-specific accelerators integrated in the same system.

6.1. Motivation

This work is at the intersection between the IoT domain and the autonomous vehicle domain. In particular, autonomous vehicles need to be able to recognize lane lines on roadways to calculate the desired trajectory and implement autonomous driving.

This translates into a line detection problem in images that is usually solved with neural networks. Now, neural networks are based on applying a matrix of weights to a matrix of data, i.e., matrix multiplication. Matrix multiplications are known to be expensive to perform, even more so when we work with high resolution images, since they offer us a large number of pixels.

Therefore, in this work, we intend to solve the problem of the large amount of computation from an architectural point of view. In particular, different homogeneous and heterogeneous architectures will be designed, consisting of the general-purpose *Rocket* and *Boom* processors and the specific-purpose *Gemmini* and *Hwacha* accelerators for matrix multiplication and vector processing, respectively.

This architecture will be designed to optimize the execution of an image line recognition application, whose execution in a real scenario would be on board on an autonomous vehicle. This time, the application will make use of known algorithms for image processing instead of implementing a neural network to simplify the code used.

Both the processors and the accelerator and vector coprocessor chosen use the RISC-V ISA along with some custom instructions for the accelerator and coprocessor. Both are chosen because RISC-V is an open source ISA, as well as the tools that are available for generating architectures and performing simulations. In adiition the architecture generation tools that will be used such as Chipyard and FireSim allow to modify architecture components in detail in a simple and comfortable way. Thus, modules can be added or removed to adapt the architecture as much as possible to the use case and achieve the desired balance between performance and consumption that is pursued in IoT.

6.2. Objectives

In the following, we will describe the objectives of this work. First, we aim to explore the RISC-V ecosystem, both the instruction ISA and the cores, the specific purpose hardware options and the software tools to generate hardware designs that are available to the user.

From the first objectives, it is necessary to explore heterogeneous architectures, to become familiar with the use of the specific instructions that are added to the RISC-V ISA with some modules and to be able to connect a processor and specific-purpose hardware in the same hardware design.

Also, it is intended to relate us to the world of autonomous vehicles, in particular, to image processing and its importance in this context. In this work, we will focus on image line detection, since it is a technology increasingly present in current vehicles as well as in autonomous vehicle prototypes. To do so, we will familiarize ourselves with the image processing algorithms used in this scenario: the Canny algorithm and the Hough transform.

All this leads us to meet our ultimate goal, which is to achieve a platform on which we

can run customized applications on architectures designed specifically for such applications and achieve in the execution of these applications a high performance at a low energy cost, since the technology embedded in an autonomous vehicle must be fast but cheap in terms of energy.

6.3. Workplan

The work plan that we are going to follow to complete the objectives that have been set out starts by making a first approach to the image processing code and detecting small improvements that can be applied in a simple way, such as using the smallest size variables that are adapted to the situation. Then, a performance study of the image processing code will be carried out by means of a *profiling* to find out in which parts it consumes more time and, therefore, more energy and resources as well. Thus, we will know which operations we have to execute at hardware level in the architecture.

Next, we will perform an exhaustive investigation of the processors, coprocessors and accelerators that currently implement the RISC-V instruction ISA and are free and open source, in order to choose those that best suit the needs of our application.

Once we have decided on the necessary hardware, we will use the Chipyard and FireSim development environments to design the architectures that we believe are most appropriate and, subsequently, generate images of those architectures to dump them onto an FPGA. In addition, we will compile the code of our main line detection application using the RISC-V cross compiler and other tools at our disposal to finally simulate the execution of the main line detection program on those architectures.

Finally, we will discuss the results obtained in the executions of different applications on different architectures in order to conclude which architecture is best suited to our use case.

The next chapter of this work will detail the main characteristics of both the chosen hardware and the development environments to be used. In subsequent chapters, the processing algorithms used and the results obtained will be presented. The work ends with the conclusions derived from the realization of the work.

Chapter 7

Conclusions and Future Work

This chapter recaps the main conclusions and also the potential lines of future work.

7.1. Conclusions

The conclusions derived from this work, from more general ones to those more particular, are the following. First, the RISC-V architecture, with its hw/sw ecosystem, reveals as an interesting solution, especially in the IoT field, due to its *opensource*, modularity and versatility features. It makes it possible to develop customized systems for different scenarios.

Second, the experimental environment employed is powerful and it allows to experiment with different configurations. However, it is significantly complex and is not well documented. An important contribution of this work is the documentation of all required steps to install and configure the environment, as well as to generate different designs and *workloads*.

From a more specific point of view, we compared the execution of different algorithms in cores with different features, such as Rocket and Boom. The results show that for some examples the execution times in the different cores are significantly different, while for other examples they are not. For the specific case of line detection algorithms, the fastest Boom processors are chosen.

We also run codes in multicore processors, verifying that the execution of tasks in the different cores was being done correctly and that the speedups with respect to the single-cores version was the one expected.

Besides, we generated designs with specific purpose accelerators (Gemmini) and vector coprocessors (Hwacha) and succedded in executing code in both.

Finally, line detection algorithms have been adapted in order to decrease their execution time without loss. Further modifications where needed for their execution in the accelerators.

The execution of line-detection algorithms on multicore systems had worked properly, and we proved that the Boom processor reports lower execution times.

However, the execution of these algorithms on a system equipped with a Gemmini accelerator did not report the expected results. From the experiment 4.6.3 we can infer that the Gemmini accelerator can not be employed in multiplications of low-dimension matrix, since a great part of its potential is clearly wasted and the performance delivered is not improved. Nevertheless, for larger matrix that we can found in IoT environments where neural networks are used, this accelerator would fulfill the program requirements and higher performance would be achieved, as shown in [16].

The execution using Hwacha does not end correctly for any of the configurations tried, due to lack of memory.

7.2. Future work

As already mentioned, in this work we have developed a methodology and documentation that will allow us to obtain new designs in a simple way. This opens new avenues to raise other problems to be solved, or to look for better solutions to the problems raised here.

In particular, we want to look for other applications that can take the advantage of the potential of the Gemmini accelerator and experiment with other designs of the Hwacha coprocessor, to try to run the line detection aplication on it.

Finally, we would like to complement the line detection application with some others, so that a more complete autonomous driving scenario could be achieved.

Bibliografía

[1] Amazon, I. Amazon web services. https://aws.amazon.com/es, 2021.

[2] Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y. S., Asanović, K. y Nikolić, B. Chipyard: Integrated design, simulation, and implementation framework for custom socs. 2020.

[3] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D. y Koenig, J. The rocket chip generator. University of California, Berkeley, April 15, 2016.

[4] Balfour, J., Dally, W., Black-Schaffer, D., Parikh, V. y Park, J. An energy-efficient processor architecture for embedded systems. 2008.

[5] Batten, C. Simplified vector-thread architectures for flexible and efficient data-parallel accelerators. 2010.

[6] Bose, P., Vega, A., Adve, S., Adve, V., Misailovic, S., Carloni, L., Shepard, K., Brooks, D., Reddi, V. J. y Wei, G.-Y. Secure and resilient socs for autonomous vehicles. 2021.

[7] Canny, J. F. Finding edges and lines in images. 1983.

[8] Chisel. Chisel/firrtl hardware compiler framework. https://github.com/freechipsproject/chisel3, 2020.

[9] Dabbelt, D., Schmidt, C., Love, E., Mao, H., Karandikar, S. y Asanović, K. Vector processors for energy-efficient embedded systems. University of California, Berkeley.

[10] developers, B. Buildroot github. https://github.com/buildroot/buildroot, 2021.

[11] developers, B. The buildroot user manual. https://buildroot.org/downloads/manual/manual.pdf, 2021.

[12] developers, F. Fedora project. https://docs.fedoraproject.org/en-US/project/, 2021.

[13] DEVELOPERS, F. Fedora quick docs. https://docs.fedoraproject.org/en-US/quick-docs/, 2021.

[14] GENC, H., HAJ-ALI, A., IYER, V., AMID, A., MAO, H., WRIGHT, J., SCHMIDT, C., ZHAO, J., OU, A., BANISTER, M., SHAO, Y. S., NIKOLIC, B., STOICA, I. y ASANOVIC, K. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. 2019.

[15] GENC, H., KIM, S., AMID, A., HAJ-ALI, A., IYER, V., PRAKASH, P., ZHAO, J., GRUBB, D., LIEW, H., MAO, H., OU, A., SCHMIDT, C., STEFFL, S., WRIGHT, J., STOICA, I., RAGAN-KELLEY, J., ASANOVIC, K., NIKOLIC, B. y SHAO, Y. S. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. 2021.

[16] GENC, H., KIM, S., AMID, A., HAJ-ALI, A., IYER, V., PRAKASH, P., ZHAO, J., GRUBB, D., LIEW, H., MAO, H., OU, A., SCHMIDT, C., STEFFL, S., WRIGHT, J., STOICA, I., RAGAN-KELLEY, J., ASANOVIC, K., NIKOLIC, B. y SHAO, Y. S. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. https://people.eecs.berkeley.edu/~ysshao/assets/papers/genc2021-dac.pdf, 2021.

[17] KARANDIKAR, S., MAO, H., KIM, D., BIANCOLIN, D., AMID, A., LEE, D., PEMBERTON, N., AMARO, E., SCHMIDT, C., CHOPRA, A., HUANG, Q., KOVACS, K., NIKOLIC, B., KATZ, R., BACHRACH, J. y Ć, K. A. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. 2018.

[18] KARANDIKAR, S., MAO, H., KIM, D., BIANCOLIN, D., AMID, A. y RESEARCH., B. A. Firesim docs. https://docs.fires.im/en/latest/FireSim-Basics.html, 2021.

[19] KESSLER, R. E. The alpha 21264 microprocessor. 1999.

[20] KIM, D., CELIO, C., BIANCOLIN, D., BACHRACH, J. y ASANOVIĆ, K. Evaluation of risc-v rtl with fpga-accelerated simulation. University of California, Berkeley.

[21] KOZYRAKIS, C. Scalable vector media-processors for embedded systems. 2002.

[22] KOZYRAKIS, C. E. y PATTERSON, D. A. Scalable, vector processors for embedded systems. 2003.

[23] KRASHINSKY, R. Vector-thread architecture and implementation. 2007.

[24] KRASHINSKY, R., BATTEN, C., HAMPTON, M., GERDING, S., PHARRIS, B., CASPER, J. y ASANOVIC, K. The vector-thread architecture. 2004.

[25] LEE, Y. Efficient vlsi implementations of vector-thread architectures. 2011.

[26] LEE, Y., AVIZIENIS, R., BISHARA, A., XIA, R., LOCKHART, D., BATTEN, C. y ASANOVIC, K. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. 2013.

[27] LEE, Y., SCHMIDT, C., OU, A., WATERMAN, A. y ASANOVIĆ, K. The hwacha vector-fetch architecture manual, version 3.8.1. technical report, ucb/eecs-2015-262. University of California, Berkeley, December 2015.

[28] Mao, H. y Zhao, J. Chipyard basics. https://fires.im/micro19-slides-pdf/02_chipyard_basics.pdf, 2019.

[29] Mao, H. y Zhao, J. Generating rocket/boom socs with rocket chip. UC Berkeley Architecture Research, Hot Chips 2019.

[30] Patterson, D. y Waterman, A. Guía práctica de risc-v: El atlas de una arquitectura abierta. primera edición, 1.0.5. http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf, 11 de Julio de 2018.

[31] Research., B. A. Firemarshal docs. https://firemarshal.readthedocs.io/en/latest/#, 2021.

[32] Russell, R. M. The cray-1 computer system. 1978.

[33] Schmidt, C. y Izraelevitz, A. A fast parameterized sha3 accelerator. https://people.eecs.berkeley.edu/~colins/papers/SHA3.pdf, 2012.

[34] Schmidt, C., Ou, A. y Asanovic, K. Hwacha v4: Decoupled data parallel custom extension. https://riscv.org/wp-content/uploads/2018/12/Hwacha-A-Data-Parallel-RISC-V-Extension-and-Implementation-Schmidt-Ou-.pdf, 2018.

[35] Waterman, A., Lee, Y., Patterson, D. y Asanovic, K. The risc-v instruction set manual, volume i: User-level isa version 2.0. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf, May 2016.

[36] Wawrzynek, J., Asanovic, K., Lazzaro, J. y Zimmer, B. Cs250 vlsi systems design. lecture 11: Patterns for communication links, rocket μarchitecture, testing. UC Berkeley, Fall 2011.

[37] Yeager, K. C. The mips r10000 superscalar microprocessor. 1996.

[38] Zhao, J., Korpan, B., Gonzalez, A. y Asanovic, K. Sonicboom: The 3rd generation berkeley out-of-order machine. 2020.

[39] Zhao, J., Korpan, B., Gonzalez, A. y Asanovic, K. Sonicboom: The 3rd generation berkeley out-of-order machine. https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf, 2020.